

国外著名高等院校
信息科学与技术优秀教材

PTR
PH

IBM PC 汇编语言 程序设计 (第五版)

IBM PC Assembly Language
and Programming
FIFTH EDITION

Peter Abel 著
沈美明 温冬婵 译



中文版 ✓

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

国外著名高等院校信息科学与技术优秀教材

IBM PC 汇编语言程序设计 (第五版)

中文版 

IBM PC Assembly Language and Programming FIFTH EDITION

在本书第五版中，作者充分考虑了各层次读者的编程需求，力图帮助读者掌握汇编语言程序设计技术，既介绍了硬件和语言的简单要素，也提供了技术细节和所需指令。

本书的内容具有以下特色：

- PC 的硬件组成。
- 机器语言代码与十六进制格式。
- 汇编、连接和执行程序的步骤。
- 用汇编语言编写程序处理键盘、屏幕与鼠标事件，进行各种数据格式之间的转换，实现表格的查找与排序，以及处理磁盘操作等。
- 作为程序调试的辅助手段，跟踪程序的执行。
- 编写宏指令，以加快编码的速度。
- 把分别汇编的程序连接成一个可执行程序。

第五版中还进行了许多更新和补充，包括视频系统组成和视频操作的详细介绍，例程的修改和增加，保护模式、堆栈用途、寻址方式和数组处理等内容。

ISBN 7-115-10352-6



9 787115 103529 >

ISBN7-115-10352-6/TP·2911

定价：48.00 元

人民邮电出版社
<http://www.ptpress.com.cn>

国外著名高等院校信息科学与技术优秀教材

IBM PC 汇编语言程序设计

(第五版)

Peter Abel 著

沈美明 温冬婵 译

人 民 邮 电 出 版 社

图书在版编目 (CIP) 数据

IBM PC 汇编语言程序设计: 第五版 / () 埃布尔 (Abel,P) 著: 沈美明, 温冬婵译. —北京: 人民邮电出版社, 2002.9

国外著名高等院校信息科学与技术优秀教材

ISBN 7-115-10352-6

I. I... II. ①埃... ②沈... ③温... III. 汇编语言—程序设计—高等学校—教材—英文
IV. TP313

中国版本图书馆 CIP 数据核字 (2002) 第 049563 号

版 权 声 明

Simplified Chinese Edition Copyright © 2002 by PEARSON EDUCATION NORTH ASIA LIMITED and POSTS & TELECOMMUNICATIONS PRESS.

IBM PC Assembly Language and Programming, Fifth Edition

By Peter Abel

Copyright © 2001

All Rights Reserved.

Published by arrangement with Prentice Hall, Pearson Education, Inc.

The edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative of Hong Kong and Macau).

本书封面贴有 **Pearson Education** 出版集团激光防伪标签, 无标签者不得销售。

国外著名高等院校信息科学与技术优秀教材 IBM PC 汇编语言程序设计 (第五版)

- ◆ 著 Peter Abel
- 译 沈美明 温冬婵
- 责任编辑 陈冀康
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
- ◆ 开本: 800×1000 1/16
印张: 32.25
字数: 783 千字 2002 年 9 月第 1 版
印数: 1-5 000 册 2002 年 9 月北京第 1 次印刷

著作权合同登记 图字: 01-2001-4826 号

ISBN 7-115-10352-6/TP · 2911

定价: 48.00 元

本书如有印装质量问题, 请与本社联系 电话: (010) 67129223

内容提要

本书详细介绍了 80x86 汇编语言程序设计的方法和技术。

全书分为 7 个部分 26 章，从硬件和软件基础知识、汇编语言基础知识、视频与键盘操作、数据操作、高级输入/输出等几个方面进行讲解和分析，阐述了宏的使用、连接、程序装入和覆盖等特殊编程技术，最后 3 章以及附录部分的内容既是知识的扩展又是实用的参考资料。书中提供了大量程序实例，除最后 3 章以外，每章还附有习题。

本书可用作高等院校及大中专院校汇编语言程序设计课程的教材或参考书，也能够帮助初学者通过自学上机掌握汇编语言程序设计的一般技术。

译者序

《IBM PC 汇编语言程序设计》一书是由英国哥伦比亚工业学院 Peter Abel 教授编写的，是国外同类著作中一本较为实用和知名的教材。这是该书经最新修订后的第五版。我们有幸把它介绍给国内广大读者，相信会对大家学习汇编语言程序设计有所帮助。

与前四版相比，本书提供了一些新的资料，特别在视频系统的组成及操作方面做了更详细的说明，对不少程序实例做了修订或补充，丰富了有关保护模式、堆栈的用法、寻址方式以及数组处理等方面的内容。全书内容全面，并以程序设计方法和技术为线索，由浅入深，循序渐近地进行全书的组织。因此，本书更适合于读者通过自学上机的方式，学习和掌握汇编语言程序设计技术。同时，本书也可以作为大学或大、中专院校汇编语言程序设计课程的教材或教学参考书。

本书的第一、二、四、六部分，第七部分的第 25、26 章及附录 A~E 由沈美明翻译；第三、五部分，第七部分的第 24 章及附录 F 由温冬婵翻译。

由于译者水平有限，译文难免会有不当或欠妥之处，敬请读者指正。

译者
2002 年 4 月

个人计算机的核心是微处理器，它负责处理计算机在算术运算、逻辑运算和控制方面的需求。微处理器起源于 20 世纪 60 年代，当时的研究设计人员所设计的集成电路（IC）是在一块硅“芯片”上把各种电子元件组合成一个部件。20 世纪 70 年代初期，Intel 公司制造出了 8008 芯片，它宣告了第一代微处理器的诞生。

到了 1974 年，8008 已发展到 8080，这是一种通用的、流行的第二代微处理器。1978 年，Intel 又生产出第三代的 8086 微处理器，在设计上取得了重要的进展。8088 是 8086 的改型，它提供一种稍微简单的设计，以及与当时流行的输入/输出设备的兼容性。1981 年 IBM 公司选中 8088 用于它即将推出的个人计算机中。8086 的各种改进版包括 80286，80386，80486，Pentium，其他更先进的 Pentium，以及 Celeron 等型号，它们当中的每一种都提供增强了的处理能力。

每种处理器系列都有它自己唯一的指令系统，它们用于控制处理器的操作。例如，从键盘接受输入，在屏幕上显示数据，以及完成算术运算。这种指令系统（机器语言）非常复杂，在用于程序开发时又令人很难理解。软件供应商为处理器系列提供了一种汇编语言，它以较易于理解的符号代码来表示各种不同的指令。

程序设计的级别

程序设计语言的级别如下：

- 机器语言，它由处理器每次执行一条的各个指令组成；它们被嵌入到操作系统和机器体系结构的低层只读部分。
- 低级汇编语言，它是为专门的处理器系列设计的；这种符号指令直接与机器语言指令一一对应并被汇编成机器语言。
- 高级语言，例如 C、C++ 以及 Visual BASIC。它们被设计成与特定计算机的技术细节无关的语言，用高级语言编译的语句一般会产生许多低级指令。

汇编语言的优点

了解和使用汇编语言有以下一些优点：

- 说明程序是如何与操作系统、处理器和 BIOS 接口的。
- 说明数据是如何表示并存储在存储器与外部设备上的。
- 阐述处理器是如何访问与执行指令的，以及指令是如何访问与处理数据的。
- 阐述程序是如何访问外部设备的。

另外，使用汇编语言的理由是：

- 用汇编语言编写的程序比用高级语言编写的程序所要求的存储空间与执行时间将显著减少。
- 汇编语言使程序员可以完成技术性非常强的任务；而使用高级语言，即使可能做到，恐怕也会非常困难。
- 虽然大多数软件专家是用高级语言开发新的应用软件的（那样做，程序会比较容易编写和维护），但是对于执行时间要求比较苛刻的部分，常常还要用汇编语言来重新编写。
- 常驻程序（当其他程序运行的时候，常驻在存储器中的程序）和中断服务例行程序（处理输入与输出的程序）几乎都是用汇编语言开发的。

学习汇编语言需要以下工具：

- IBM 个人计算机（任何一种型号）或等效的兼容机。
- 一种 Windows 95/98 或 DOS 操作系统并熟悉它的使用。当工作在 DOS 这种相对简单的操作系统而不是在 Windows 环境时，学习汇编语言的难度要小一些。你可以先在 DOS 中练习，以后再进入到 Windows 环境中。
- 一种汇编翻译程序。一般的供应商有 Microsoft, Borland 以及 SLR 系统。

学习汇编语言不要求以下知识：

- 程序设计语言的知识，尽管这些知识可能有助于你更快地理解一些编程概念。
- 电子或电路方面的知识。这本书提供了汇编语言程序设计所要求的有关 PC 体系结构的所有信息。

本书的焦点

为了帮助读者学习汇编语言程序设计，本书首先涉及的是比较简单的硬件和语言方面的情况，然后介绍所需要的指令。同时，本书还力图使程序示例更为清晰。因此，程序使用了那些最容易理解的指令和方法，尽管专业程序员通常会用较为复杂而又不太清晰的代码来解决类似的问题。

程序还省略了宏指令（将在第 21 章介绍）。虽然专业程序员广泛地使用宏，但它们在书中的出现会影响对语言原理的学习。你一旦学会了这些原理，这些专业技能自然就会得到提高。

学习的方法

本书可以起到指导与参考两方面的作用。为了能最有效地利用自己在 PC 和软件上的投

资，你应该仔细地把每一章读一遍并重读那些没有马上搞清楚的内容。利用所举的程序例子并把它们放在你的计算机上加以执行（或“运行”）。同时，一定要做每章后面的习题。

前面 8 章提供的是汇编语言的一些基本知识。在学完这些章节之后，你可以继续学习第 9, 11, 12, 14, 15, 16, 20, 21 或 22 章。24 章到 26 章作为参考。相互关联的章节是：

- 第 8 章到第 10 章（屏幕和键盘操作）。
- 第 12 章和第 13 章（算术运算操作）。
- 第 16 章到第 19 章（磁盘处理）。
- 第 22 章和第 23 章（子程序和覆盖）。

在学完这本书的时候，你应该能够：

- 了解个人计算机的硬件。
- 了解机器语言代码和十六进制格式。
- 了解汇编、连接与执行程序的步骤。
- 用汇编语言编写管理键盘与屏幕，完成算术运算，ASCII 与二进制格式之间的转换，实现表格的查找与排序，以及处理磁盘输入与输出的程序。
- 作为程序调试的一种辅助手段，跟踪机器的执行过程。
- 编写你自己的宏指令，以便提高编码速度。
- 连接各个汇编语言程序，形成一个可执行程序。

学习汇编语言并使你的程序正确运行是一种令人兴奋而又有挑战性的体验。对于所付出的时间和精力，你肯定会得到很大的回报。

有关第五版的说明

第五版在以下几方面内容上比以前版本大有提高：

- 更多的 Intel Pentium 处理器的特性。
- 更多的程序举例和习题。
- 更早地引入中断操作。
- 全文重组和注释的修订。
- 更多的有关保护方式、传送参数、堆栈使用、寻址方式、显示系统与 INT 10H 功能、数组处理、子程序以及端口方面的内容。
- 修订并增加了每章末尾的习题。

第四版的用户应该注意第 7 章的内容（.COM 程序）已和这一版的第 5 章合并在一起了。另外，第 21 章是以这样的方法分散的：有关鼠标处理的内容单独成为一章（第 15 章），第 15 章的其余内容（端口、串输入/输出以及发声）与 BIOS 和程序中断合并在一起，放在第 24 章。

学生或读者注意：可以在网上查阅所选择习题的答案，下载书中的程序，以及有关实验方面的问题。网址是 www.prenhall.com/abel。

感谢

作者感谢所有对本书提供过帮助与合作的人，他们曾为本书先前的版本提出过建议，进行了文稿的审阅和校对。

基础的重要性(程序员之路)

学习编程有几年了,感觉走了不少弯路,而不少的学弟学妹又在重蹈我当初的覆辙,不免有些痛心。最近在网上也看了许多前辈们的经验建议,再结合自己的学习经历在这里谈谈基础的重要性,帮助大家少走些弯路。

什么是基础呢?就是要把我们大学所学的离散数学,算法与数据结构,操作系统,计算机体系结构,编译原理等课程学好,对计算机的体系,CPU本身,操作系统内核,系统平台,面向对象编程,程序的性能等要有深层次的掌握。初学者可能体会不到这些基础的重要性,学习jsp,donet,mfc,vb的朋友甚至会对这些嗤之以鼻,但是一开始没学好基础就去学jsp或donet会产生很坏的影响,而且陷入其中不能自拔。

我上大二的时候还对编程没什么概念,就上了门C++也不知道能干什么,老师说MFC也不知道是什么东西,看别的同学在学asp.net就跟着学了,然后就了解到.net,j2ee,php是什么了,就觉得软件开发就是用这些了,而上的那些专业课又与我们学的sqlserver啊,css啊,ajax啊,毫无关系,就感慨啊,还不如回家自学去就为一个文凭吗?还不如去培训,浪费这么多钱.于是天天基本上没去上什么课,天天就在做网站,几个学期就做了三个网站。感觉做这些网站就是学到些技巧,没什么进步,这些技巧就好比别人的名字,告诉你你就知道了,网上也都可以搜到。那时候就觉得把.net学好就行了,搞j2ee的比较难,搞api编程就别想了,操作系统更是望尘莫及了。后来随着学习的深入和看了网上许多前辈们的建议才对这些基础的重要性有所体会。

虽然.net或java的开发并不直接用到汇编,操作系统这些,但是不掌握这些基础是有很大问题的,因为你只知其然不知其所有然,在mfc和.net里面控件一拖什么都做好了,很方便,但是出了问题可能就解决不了,有些在网上搜都搜不到。这就是基础没打好,不知道它的原理就不知道出错的原因。在学.net的时候常会讨论那些控件该不该用别人说尽量别用也不知道为什么?不让用是因为你在高层开发,你不知道它的原理出错了你可能解决不了,但其实是应该用的,不然人家开发它干嘛,但要在了解它的原理后去用就会很方便。

要编写出优秀的代码同样要扎实的基础,如果数据结构和算法学的不好,怎么对程序的性能进行优化,怎样从类库中选择合适的数据结构。如果不了解操作系统,怎样能了解这些开发工具的原理,它们都是基于操作系统的。不了解汇编,编译原理,怎么知道程序运行时要多长时间要多少内存,就不能编出高效的代码。

如果没有学好基础一开始就去学.net,java这些越往后就会觉得越吃力,它们涉及的技术太多了,而且不但在更新,对于三层啊,mvc,orm这些架构,你只会用也不明白为什么用,就感觉心里虚,感觉没学好。而你把面向对象,软件工程,设计模式这些基础学好了再去看这些就可以一不变应万变。

大家不要被新名词、新技术所迷惑.NET、XML等等技术固然诱人,可是如果自己的基础不扎实,就像是在云里雾里行走一样,只能看到眼前,不能看到更远的地方。这些新鲜的技术掩盖了许多底层的原理,要想真正的学习技术还是走下云端,扎扎实实的把基础知识学好,有了这些基础,要掌握那些新技术也就很容易了。

开始编程应该先学C/C++,系统api编程,因为它们更接近底层,学习他们更能搞清楚原理。学好了c/C++编程和基础,再去学习mfc,.net这些就会比较轻松,而且很踏实。假设学习VB编程需要4个月,学习基础课程和VC的程序设计需要1年。那么如果你先学VB,再来学习后者,时间不会减少,还是1年,而反过来,如果先学习后者,再来学VB,也许你只需要1个星期就能学得非常熟练。

编程就好比练功，如果学习.net,mfc,vb等具体的语言和工具是外功(招式)，对基础的学习就是内功,只注重招式而内功不扎实是不可能成为高手的。很多人会认为《射雕英雄传》中马玉道长什么都没有教郭靖，马道长教的表面看来是马步冲拳实则都是内功心法，郭靖拜师洪七之后开始练习降龙十八掌凭借的就是这深厚的内功，吞食蝮蛇宝血又加上练习了周博通传授的九阴真经和外加功夫双手互搏技之后，终于练就行走江湖的武功，由此可见马玉道长传授给了郭靖的是最基础的，也是最重要的观念,编程就好比盖高楼，根基没打好早晚有一天会垮掉的，而且盖得越高，损失也越惨重。这些底层知识和课本不是没有用也不是高深的不能学，而是我们必须掌握的基础。

这些是个人的愚见，说的不是很清楚，大家可以看看这些前辈们的经验，相信看完后大家一定会有所体会的。为了方便大家阅读，我把这些前辈们的建议的文章整理成了pdf,大家在下面下载吧!希望对大家有帮助。pdf地址:<http://bbs.theithome.com/read-htm-tid-123.html>

说了这么多无非是想告诫大家要打好扎实的基础，不要只顾追求时髦的技术，打好基础再去学那些技术或是参加些培训，对自身的发展会更好的。

基础这么重要怎样学好它呢？我觉得学好它们应该对照这些基础课程所涉及的方面,多看一些经典书籍,像算法导论,编程珠玑,代码大全(具体介绍在本论坛每本书的版块里)等,这些经典书籍不仅能帮助我们打好基础，而且对我们的程序人生也能产生莫大的影响，相信认真研究看完这些书籍后，我们的程序之路会十分顺畅。然而这些书籍并不好读,有些甚至相当难读，国内的大学用这些书当教材的也不多,这些书又偏向理论,自己读起来难免会有些枯燥无味。于是就想到建一个论坛，大家共同讨论学习这些书籍，就会学的更踏实更牢固更有趣,这样就能为以后的学习打下扎实的基础。

本论坛特色：bbs.theithome.com

- 1.为计算机初学者或基础不太扎实的朋友指明方向，要注重内功
- 2.为学习者推荐经典书籍，指明应看哪些书籍，怎样练内功
- 3.为学习者提供一个交流的地方，更容易学好，不会那么枯燥
- 4.对每本书分章分别讨论，更专，会学的更踏实更牢固
- 5.讨论的都是经典书籍，每一本都会让我们受益匪浅,对每本书分别讨论是很有意义的。

1. 计算机科学概论

计算机科学概论

2. 计算机数学基础

高等数学

线性代数

概率论与数理统计

离散数学及其应用

离散数学教程(北大版)

什么是数学

具体数学: 计算机科学基础

3. C语言

谭浩强C程序设计

C primer plus

The C programming language

C和指针

C专家教程

C陷阱与缺陷

c语言解惑

C标准库

你必须知道的495个C语言问题

4. 算法与数据结构

数据结构(清华版)

数据结构与算法分析—C语言描述

编程珠玑

编程珠玑II

算法导论

计算机程序设计艺术卷1

计算机程序设计艺术卷2

计算机程序设计艺术卷3

5. 电子技术基础

模拟电子技术(童诗白版)

数字逻辑与数字集成电路(清华版)

6. 汇编语言

汇编语言(王爽版)

80X86汇编语言程序设计教程

Intel汇编语言程序设计

IBM PC汇编语言程序设计(国外版)

高级汇编语言程序设计

保护方式下的80386及其编程

黑客反汇编揭秘

Windows环境下32位汇编语言程序设计

7. 计算机硬件原理

计算机组成-结构化方法

微机原理与接口技术(陈光军版)

计算机体系结构(张晨曦版)

计算机组成与设计硬件/软件接口

Intel微处理器结构、编程与接口

计算机体系结构(量化研究方法)

编程卓越之道卷1

编程卓越之道卷2

深入理解计算机系统

编码的奥秘

8. 数据库系统原理

数据库系统概念

数据库系统导论

数据库系统实现

9. 编译原理

编译原理(清华第2版)

编译原理及实践

编译原理: 原则、技术和工具

现代编译原理-C语言描述

高级编译器设计与实现

10. 操作系统原理

操作系统概念

现代操作系统

链接器和加载器

程序员的自我修养: 链接、装载与库

自己动手写操作系统

操作系统设计与实现

11. 计算机网络

计算机网络(Computer Networks)

TCP-IP详解卷1

TCP-IP详解卷2

TCP-IP详解卷3

用TCP/IP进行网际互联(第一卷)

用TCP/IP进行网际互联第二卷

用TCP/IP进行网际互联第三卷

12. 软件工程和面向对象程序设计

C++编程思想卷1

java编程思想

软件工程(Software.Engineering)

软件工程: 实践者的研究方法

深入浅出面向对象分析与设计

head first设计模式

道法自然: 面向对象实践指南

面向对象分析与设计

敏捷软件开发: 原则、模式与实践

设计模式: 可复用面向对象软件的基础

测试驱动开发

重构—改善既有代码的设计

代码大全

程序设计实践

程序员修炼之道: 从小工到专家

卓有成效的程序员

代码之美

人月神话

计算机程序的构造和解释

观止-微软创建NT和未来的夺命狂奔

代码优化: 有效使用内存[美]克里斯·卡斯基

编程高手箴言(梁肇新)

游戏之旅-我的编程感悟(云风)

13. windows编程基础

Windows操作系统原理

Inside Windows 2000

深入解析Windows操作系统

天书夜读: 从汇编语言到Windows内核编程

windows程序设计

WINDOWS核心编程

14. linux/unix编程基础

鸟哥的Linux私房菜: 基础学习篇

鸟哥的Linux私房菜: 服务器架设篇

linux程序设计

UNIX环境高级编程

Unix网络编程卷1

UNIX网络编程卷2

UNIX编程艺术

UNIX Shell范例精解

15. Linux/unix内核源代码和驱动程序

Linux内核设计与实现

LINUX内核源代码情景分析

深入理解LINUX内核

Linux内核完全注释

Linux设备驱动程序

16. C++语言

C++编程思想2

Essential C++

C++ primer

C++程序设计语言

C++语言的设计和演化

Accelerated C++

Effective C++

More Effective C++

Exceptional C++

More Exceptional C++

C++设计新思维

深度探索C++对象模型

C++沉思录

C++ Templates: The Complete Guide

C++ FAQs

17. 标准库STL使用

C++标准程序库

Effective STL

泛型编程与STL

18. STL源代码

STL源码剖析

19. java语言

java编程思想

Java编程规范

3.4 机器语言举例 1: 使用立即数据	35
3.5 机器语言举例 2: 使用定义的数据	38
3.6 一个汇编语言程序	41
3.7 使用 INT 指令	42
3.8 使用 PTR 操作符	45
3.9 要点	46
3.10 习题	46

第二部分 汇编语言的基础知识

第 4 章 汇编语言编码要求	51
----------------------	----

4.1 引言	51
4.2 汇编语言特性	52
4.3 常规的段伪操作	57
4.4 简化的段伪操作	60
4.5 保护模式下的初始化	62
4.6 定义数据类型	62
4.7 相等伪操作	67
4.8 要点	68
4.9 习题	69

第 5 章 汇编、连接与执行程序	71
------------------------	----

5.1 引言	71
5.2 为汇编与执行准备程序	71
5.3 二遍扫视汇编程序	77
5.4 连接目标程序	77
5.5 执行程序	79
5.6 交叉引用表	79
5.7 出错诊断	80
5.8 汇编程序位置计数器	81
5.9 编写.COM 程序	81
5.10 要点	84
5.11 习题	85

第 6 章 符号指令与寻址	87
---------------------	----

6.1 引言	87
6.2 符号指令系统——概述	87
6.3 数据传送指令	90
6.4 基本算术指令	92

6.5 重复传送操作	93
6.6 INT 指令	94
6.7 寻址方式	95
6.8 段跨越前缀	98
6.9 近地址与远地址	99
6.10 对齐数据地址	99
6.11 要点	100
6.12 习题	100
第 7 章 程序逻辑与控制	103
7.1 引言	103
7.2 短地址, 近地址和远地址	103
7.3 JMP 指令	104
7.4 LOOP 指令	106
7.5 标志寄存器	107
7.6 CMP 指令	108
7.7 条件转移指令	109
7.8 调用过程	112
7.9 程序执行对堆栈的影响	114
7.10 布尔操作	117
7.11 移位	120
7.12 循环移位	123
7.13 组织一个程序	125
7.14 要点	126
7.15 习题	126

第三部分 视频与键盘操作

第 8 章 视频和键盘处理入门	131
8.1 引言	131
8.2 屏幕特征	131
8.3 设置光标	132
8.4 清屏	132
8.5 屏幕显示的 INT 21H 功能 09H	133
8.6 键盘输入的 INT 21H 功能 0AH	135
8.7 屏幕显示的 INT 21H 功能 02H	140
8.8 文件代号	140
8.9 屏幕显示的 INT 21H 功能 40H	141
8.10 键盘输入的 INT 21H 功能 3FH	142

8.11 要点	143
8.12 习题	144
第 9 章 视频系统	145
9.1 引言	145
9.2 视频系统的构成	145
9.3 视频方式	147
9.4 属性	148
9.5 BIOS INT 10H 操作	149
9.6 使用图形方式	165
9.7 直接视频显示	169
9.8 用于方框和菜单的 ASCII 字符	171
9.9 要点	172
9.10 习题	173
第 10 章 键盘操作	174
10.1 引言	174
10.2 BIOS 键盘数据区	175
10.3 键盘输入的 INT 21H 操作	176
10.4 键盘输入的 INT 16H 操作	177
10.5 扩展功能键和扫描码	179
10.6 BIOS INT 09H 和键盘缓冲区	183
10.7 要点	187
10.8 习题	187

第四部分 数据操作

第 11 章 处理串数据	191
11.1 引言	191
11.2 串操作的特点	191
11.3 MOVS: 串传送指令	193
11.4 LODS: 从串取指令	194
11.5 STOS: 存入串指令	195
11.6 程序: 使用 LODS 和 STOS 编辑数据	195
11.7 CMPS: 串比较指令	199
11.8 SCAS: 串扫描指令	200
11.9 串指令的另一种编码	201
11.10 复制一种模式	202
11.11 要点	203

11.12 习题	203
第 12 章 算术运算 I：处理二进制数据	205
12.1 引言	205
12.2 处理无符号与带符号的二进制数据	205
12.3 二进制数据的加法与减法	207
12.4 二进制数据乘法	211
12.5 二进制数据除法	217
12.6 数值数据处理器	222
12.7 要点	223
12.8 习题	224
第 13 章 算术运算 II：处理 ASCII 和 BCD 数据	226
13.1 引言	226
13.2 十进制格式的数据	226
13.3 处理 ASCII 数据	227
13.4 处理压缩的 BCD 数据	232
13.5 ASCII 数据转换成二进制格式	234
13.6 二进制数据转换成 ASCII 格式	235
13.7 乘积的移位与舍入	236
13.8 要点	241
13.9 习题	242
第 14 章 定义与处理表格	243
14.1 引言	243
14.2 定义表格	243
14.3 表格项目的直接寻址	245
14.4 查找表格	247
14.5 XLAT(换码)指令	251
14.6 表格项目排序	254
14.7 地址表	257
14.8 二维数组	258
14.9 要点	260
14.10 习题	260

第五部分 高级输入/输出

第 15 章 使用鼠标的设备	265
15.1 引言	265

15.2	基本的鼠标操作	266
15.3	程序：显示鼠标位置	268
15.4	更高级的鼠标操作	270
15.5	程序：按菜单使用鼠标	273
15.6	要点	275
15.7	习题	276
第 16 章	磁盘存储 I：组织方式	277
16.1	引言	277
16.2	磁盘存储设备的特征	277
16.3	磁盘系统区和数据区	280
16.4	引导记录	281
16.5	目录	282
16.6	文件分配表	283
16.7	处理磁盘文件	287
16.8	重点	288
16.9	习题	288
第 17 章	磁盘存储 II：写文件和读文件	290
17.1	引言	290
17.2	ASCII 串	290
17.3	文件代号	291
17.4	错误返回码	291
17.5	文件指针	292
17.6	建立磁盘文件	292
17.7	读磁盘文件	296
17.8	随机处理	299
17.9	要点	306
17.10	习题	306
第 18 章	磁盘存储 III：支持磁盘和文件的 INT 21H 功能	308
18.1	引言	308
18.2	处理磁盘驱动器的操作	309
18.3	处理目录和 FAT 的操作	318
18.4	处理磁盘文件的操作	320
18.5	要点	328
18.6	习题	328
第 19 章	磁盘存储 IV：INT 13H 磁盘功能	330

19.1	引言	330
19.2	BIOS 状态字节	330
19.3	基本的 INT 13H 磁盘操作	331
19.4	其他 INT 13H 磁盘操作	335
19.5	要点	339
19.6	习题	339
第 20 章	打印程序	341
20.1	引言	341
20.2	普通打印机控制符	341
20.3	INT 21H 的功能 40H: 打印字符	342
20.4	专用打印机控制符	348
20.5	BIOS INT 17H 打印功能	349
20.6	要点	350
20.7	习题	350

第六部分 特殊的课题

第 21 章	定义与使用宏	355
21.1	引言	355
21.2	简单的宏定义	356
21.3	在宏中使用参数	357
21.4	在宏中使用注释	358
21.5	嵌套的宏	360
21.6	宏伪操作	360
21.7	要点	368
21.8	习题	368
第 22 章	连接到子程序	369
22.1	引言	369
22.2	段伪操作	370
22.3	段内调用	371
22.4	段间调用	372
22.5	EXTRN 与 PUBLIC 属性	372
22.6	用 EXTRN 与 PUBLIC 作为入口点	374
22.7	代码段定义为 PUBLIC	376
22.8	使用简化段伪操作	378
22.9	传送参数到子程序	379
22.10	ENTER 与 LEAVE 指令	382

22.11	C/C++ 程序与汇编语言程序的连接	384
22.12	要点	387
22.13	习题	387
第 23 章	程序装入与覆盖	389
23.1	引言	389
23.2	程序段前缀	389
23.3	高端存储区	392
23.4	存储器分配策略	392
23.5	程序的装入程序	394
23.6	分配与释放存储器	397
23.7	装入或执行程序功能	399
23.8	程序覆盖	401
23.9	常驻程序	405
23.10	要点	408
23.11	习题	408

第七部分 参考章节

第 24 章	BIOS 数据区、中断和端口	413
24.1	引言	413
24.2	引导过程	413
24.3	BIOS 数据区	413
24.4	中断服务	417
24.5	BIOS 中断	419
24.6	INT 21H 服务例程	421
24.7	端口	424
24.8	串输入/输出	426
24.9	产生声音	427
24.10	要点	428
24.11	习题	429
第 25 章	操作符与伪操作	430
25.1	引言	430
25.2	类型区分符	430
25.3	操作符	430
25.4	伪操作	434
第 26 章	PC 指令系统	448

26.1 引言.....	448
26.2 寄存器表示法.....	448
26.3 寻址方式字节.....	449
26.4 指令系统.....	451
附录 A 十六进制数与十进制数之间的转换	477
附录 B ASCII 字符码.....	480
附录 C DEBUG 程序	482
附录 D 保留字	488
附录 E 汇编与连接程序	490
附录 F 键盘扫描码和 ASCII 码	495

第一部分

PC 硬件与软件的 基础知识

目的：说明微型计算机硬件的基本特征和程序组织

1.1 引言

用汇编语言编写一个程序，需要计算机硬件(或体系结构)方面的知识和有关指令系统的详细说明。本章提供的是关于基本硬件——位、字节、寄存器、处理器以及数据总线的说明。本书的其余部分阐述指令系统及其使用方法。

计算机主要的内部硬件部件是处理器、存储器和寄存器(寄存器是用来保存地址和数据的专用处理器部件)。外部硬件部件包括计算机的输入/输出设备，如键盘、显示器、磁盘和 CD-ROM。软件包括操作系统、各种程序以及存储在磁盘上的数据文件。

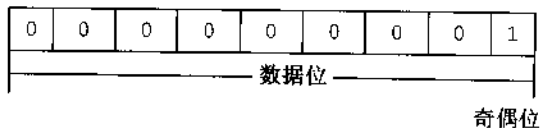
为了执行(或运行)一个程序，系统要把这个程序从外部设备复制到内部存储器(例如，人们在谈到自己的计算机有 64MB 的存储器时，指的就是内存储器或 RAM)中。处理器执行程序指令，而它的寄存器则要处理所要求的算术运算、数据传送与寻址。

1.2 位与字节

计算机存储的基本构造单元是“位”(bit)。一个位可能是“关闭”(off)态，它的值被看作是 0；或者它是“开通”(on)态，它的值则被看作是 1。单个的位提供不了更多信息，但一些位要是组合在一起就会有意想不到的作为。

字节

一组 9 个相关的位称为**字节**，它代表内存储器和外部设备的一个存储单元。每个字节由 8 个数据位和 1 个奇偶位组成：



8 个数据位为二进制运算和诸如字母 A 与星号(*)这样一些字符的表示打下了基础。字节中的 8 位允许有 $256(2^8)$ 个“开”——“关”状态的不同组合,从所有都为“关”状态(00000000)到所有都为“开”状态(11111111)。例如,字母 A 的位表示是 01000001,而星号的位表示则是 00101010,至于这些位的值是多少并不需要你去记它。

根据奇偶校验规则,在每个字节中,1 的个数必须总是奇数。因为字母 A 中有两个位值是 1,所以处理器按奇偶校验规则自动地把奇偶位设置为 1(01000001-1)。类似地,由于星号包含 3 个 1,处理器根据奇偶校验规则就把奇偶位置为 0(00101010-0)。当指令访问内存储器中的一个字节时,处理器要检查这个字节的奇偶性。如果奇偶性是偶数,系统便会假定“丢”了一位并显示出错信息。奇偶错可能是由硬件故障或是电气干扰造成的,总之是偶发的事件。

怎么做才能使计算机“知道”位值 01000001 是代表字母 A 呢?当你在键盘上键入 A 时,系统传送这个特定键盘信号到存储器并设置一个字节(在你的程序的一个单元内),其位值为 01000001。你可以按需要在存储器范围内传送该字节的内容,还可以把字母 A 打印出来或显示在屏幕上。

处理奇偶校验是一种自动的硬件功能,我们不必去管它。一个字节中的各位从右到左是按 0 到 7 编号的,如字母 A 表示如下:

位内容(A):	01000001
位编号:	76543210

相关字节。程序可以把一个或多个相关字节的组看作是一个数据单元,比如时间或距离。定义成特定值的一组字节通常是作为**数据项**或**字段**来识别的。处理器还支持一些特定的数据大小:

字。2 字节(16 位)数据项。

双字。4 字节(32 位)数据项。

四字。8 字节(64 位)数据项。

小段。16 字节(128 位)区。

千字节(KB)。数 2^{10} 等于 1 024(正好是 K 值)。因此,640K 就是 $640 \times 1\,024 = 655\,360$ 字节。

兆字节(MB)。数 2^{20} 等于 1 048 576,MB 是为 1 兆字节。

在一个字中,各位从右到左的编号是 0 到 15。对于字母 PC 则表示成最左边的字节为 P(01010000),而最右边的字节为 C(01000011):

位内容(PC):	0 1 0 1 0 0 0 0	0 1 0 0 0 0 1 1
位编号:	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

在存储器中的每个字节都有一个唯一的地址。第一个字节在最低的存储器单元中,编号为 0,第二个字节编号为 1,等等。

1.3 二进制数系统

由于计算机只能辨别 0 位和 1 位,所以它只能工作在以 2 为基数的称为二进制的计数系统中。实际上,二进制位“bit”(“位”)是“Binary digIT”的缩写。

位的集合可以表示任何数字的值。二进制数的值是由为 1 的位及其相关位置所决定的。如同十进制数一样，位置代表从右到左幂次的上升(但这是 2 的幂，不是 10 的幂)。在以下的 8 位数中，所有位均置为 1：

位的值:	1	1	1	1	1	1	1
位置的值:	128	64	32	16	8	4	2
位编号:	7	6	5	4	3	2	1

最右边的位设其值为 $1(2^0)$ ，向左的下一位设其值为 $2(2^1)$ ，再下一位的值为 $4(2^2)$ ，以此类推。在这种情况下，二进制数的值是 $1+2+4+8+16+32+64+128=255$ (或 2^8-1)。

让我们考虑一个二进制数 01000001 的值：

位的值:	0	1	0	0	0	0	0	1
位置的值:	128	64	32	16	8	4	2	1

你可以计算它的值 $1+64=65$ 。但是你可能会想到位值 01000001 也是字母 A。的确，01000001 既可以代表 65 这个数，也可以代表字母 A：

- 如果程序是为算术运算目的而定义与使用数据的，那么位值 01000001 就是表示一个二进制数，它等于十进制数 65。
- 如果程序是为描述目的而定义与使用数据的，如标题，那么 01000001 就代表一个字母字符。

当你开始编程的时候，由于要为一个特定目的而定义与使用每个数据项(即为运算目的而用的运算数据以及为显示输出而用的描述数据)，所以你就会更清楚地看到这种区别。实际上，两种用法很少会发生混淆。

二进制数不受 8 位限制。使用 16 位(或 32 位)体系结构的处理机会自动处理 16 位(或 32 位)数。对于 16 位机，所能提供的值在 $2^{16}-1$ (即 65535)之内；而对于 32 位机，所能提供的值可以达到 $2^{32}-1$ 即(4294967295)。

二进制算术运算

因为微计算机只以二进制格式完成算术运算，所以一个汇编语言程序员必须熟悉二进制格式与二进制加法。以下 4 个例子可以简单说明二进制加法：

0	0	1	1
+0	+1	+1	+1
0	1	10	11

最后两个例子表明有一个 1 的进位进到下一个(左边)位置。现在，让我们把位值 01000001 和 00101010 相加。我们是把字母 A 和一个星号相加吗？不是的，这时它们代表的是十进制值 65 和 42。

十进制	二进制
65	01000001
+ 42	+ 00101010
107	01101011

为了检验二进制和 01101011 实际上就是 107，可以把是 1 的位的值相加。作为另一个例子，我们把十进制值 60 与 53 以及它们的二进制等效值相加：

十进制	二进制
60	00111100
+ 53	+ 00110101
113	01110001

再一次检查二进制的和，实际上就是 113。

负二进制数。带符号的二进制数(即一种用于算术运算的数)是这样认定的：如果它的最左边的位是 0，它就是正数；反之，带符号的负二进制数的最左边的位是 1。但是，表示一个二进制数是负的，不能简单地把最左边的位设置成 1 就行了，如把 01000001(+65)变成 11000001，而应该把负二进制值用二进制补码表示法来表示。也就是说，把一个二进制数表示成负数的规则是：位值求反并加 1。例如，利用这一规则求出 01000001(或 65)的二进制补码：

数+65:	01000001
按位求反:	10111110
加 1:	+ 1
数-65:	10111111

当带符号的二进制数最左边的位是 1 时，它是负数，并且处理器会对它做相应的处理。但是，如果你想用加 1 的办法求得二进制数 10111111 的十进制值，你将不会得到 65。为了求得一个负二进制数的绝对值，需要采用二进制补码规则，也就是按位求反再加 1：

数 -65	10111111
按位求反:	01000000
加 1:	+ 1
数 +65	01000001

为了说明这种做法是正确的，+65 与 -65 的和应该是零。让我们试一下：

+ 65	01000001
+ (-65)	+10111111
00	(1)00000000

注意，8 位数以外还有一位溢出的 1。在这个和中，8 位的值是全 0，而在左边溢出的 1 则被丢弃了。但是，因为既有进入符号位的进位，又有进位输出，所以认为结果是正确的。

为了处理二进制减法，需要把进行减法的数转换成二进制补码格式，再把它们相加。例如，65 减 42。42 的二进制表示是 00101010，它的二进制补码是 11010110；把 -42 加上 65 就变成这样：

65	01000001
+ (-42)	+ 11010110
23	(1)00010111

结果 23 是正确的。注意：又有一个有效的进位输入和符号位输出。

假如对于二进制补码表示法还没有弄清楚,可以考虑以下问题:你必须把一个什么样的值加到二进制数 00000001 上,才会使它等于 00000000 呢?对于十进制数而言,答案将是一1。00000001 的二进制补码是 11111111,所以 +1 与 -1 相加等于零:

$$\begin{array}{r}
 1 \\
 + (-1) \\
 \hline
 \text{结果:}
 \end{array}
 \qquad
 \begin{array}{r}
 00000001 \\
 + 11111111 \\
 \hline
 (1)00000000
 \end{array}$$

忽略进位的 1,你会看到二进制数 11111111 等于十进制数 -1。你还会看到二进制数按值减少的一种模式:

+3	00000011
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110
-3	11111101

实际上,在负二进制数中为 0 的位可以指出这个数的绝对值:把每个是 0 的位的位置值看成是 1,把这些值求和并加 1。

当你接触到有关算术运算的第 12 章和第 13 章时,你将会发现有关二进制算术运算和负数的更为详细的内容。

1.4 十六进制表示法

尽管一个字节可以包含 256 种位的任意组合,但它们当中许多是无法按标准的 ASCII 字符显示或打印的。这类字符的例子包括这样一些操作的位组合,比如制表符(Tab)、回车符(Enter)、换页符(Form Feed)以及换码符(Escape)。因此,计算机设计人员开发了一种表示二进制数据的简化方法,把每个字节分成两半并把每半个字节的值表示出来。

假设你想看看存储器中四个邻接字节(一个双字)的二进制值的内容。考虑以下四个字节,它们分别表示为二进制与十进制格式:

二进制:	0101 1001	0011 0101	1010 1001	1100 1110
十进制:	5 9	3 5	10 9	12 14

由于十进制数 10, 12 和 14 每个都需要 2 个数字,让我们采用扩充记数系统,使 10=A, 11=B, 12=C, 13=D, 14=E, 15=F。这样,记数系统就包括了“数字”0 到 F,由于有 16 个这样的数字,所以这一系统便被称为十六进制(或 hex)表示法。下面就是修正过的简化数,它表示出了上面所给字节的内容:

二进制:	0101 1001	0011 0101	1010 1001	1100 1110
十六进制:	5 9	3 5	A 9	C E

图 1-1 给出了十进制数 0 到 15 及与其等值的二进制值与十六进制值。

二进制	十进制	十六进制	二进制	十进制	十六进制
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

图 1-1 二进制，十进制及十六进制表示法

汇编语言大量使用十六进制格式。一个汇编程序的列表文件中，所有地址、机器码指令以及数据常数的内容都是以十六进制格式表示的。为了调试程序，你可以使用 DEBUG 程序，它也是按十六进制格式显示地址与字节内容的。

你很快就能按十六进制格式工作。要记住：十六进制数中紧跟着十六进制 F 的是十六进制的 10，它的十进制值是 16。以下是十六进制算术运算的一些简单例子：

7	6	F	F	10	38	FF
+3	+7	+1	+F	+30	+18	+1
A	D	10	1E	40	50	100

还要注意：十六进制 40 等于十进制的 64，十六进制 100 是十进制的 256，而十六进制的 1000 则是十进制的 4096。例如 38+18=50，注意十六进制的 8+8 等于 10。

为了指明在汇编语言程序中的十六进制数，在这个数的后面你要写上一个“H”，如 25H(十进制的 37)。汇编语言要求十六进制数永远是用十进制数字 0~9 作为开始，所以就要把 B8H 写成 0B8H。在本书中，十六进制的值是用数的前面放一个“hex”或者数后面跟一个“H”来表示的(如 hex 4C 或 4CH)；二进制的值用放一个“binary”在数的前面或者数的后面跟一个“B”来表示(如 binary 01001100 或 01001100B)；而十进制值就简单地用一个数表示(如 76)。如果根据上下文能够明确地确定该基数值是十进制还是十六进制，也可以不必像上述那样标得那么清楚。

附录 A 给出了如何把十六进制数转换成十进制数格式，或者进行相反转换的说明。

1.5 ASCII 码

PC 中的数据可以按数(用于算术运算的二进制数据)或字母数字(字符与描述性数据)来分类。字母类型的数据通常用于键盘输入，屏幕与打印机输出。为了使数据表示标准化，微机设计人员采用 ASCII(美国标准信息交换代码)码。

由于一个字节包含 8 位，所以 PC 使用 8 位 ASCII 码，它可以提供 2⁸ 或 256 个字符，其中许多字符是用在键盘上的。例如，你已经见过的字母 A 的 ASCII 码：01000001(hex41)。幸运的是，你不必记住这些 ASCII 码；当需要的时候，你可以参阅附录 B 给出的完整的表，而且第 8 章还说明了在屏幕上如何把它们当中的大多数显示出来。

1.6 PC 的组成

PC 的主要组成部分是它的**系统板**(或称**主板**)。它包括处理器、协处理器、主存储器、接插件以及为可选电路卡用的扩展槽。这些槽和接插件为下列部件提供了出入口, 比如只读存储器(ROM)、随机存取存储器(RAM)、硬盘、CD-ROM 设备、附加的主存储器、显示设备、键盘、鼠标、并行与串行设备、声音合成器以及高速缓冲存储器等。处理器利用高速缓冲存储器, 减少对较低速主存储器的访问。

总线(BUS)用一些附加在系统板的线把各个部件连接起来。它在处理器、存储器和外部设备之间传送数据, 有效地处理数据流量。例如当程序需要从外部设备读取数据时, 处理器确定已送达的数据要存放在存储器的哪个地址中, 并把该地址放到地址总线上。然后, 存储器部件把数据放到数据总线上, 并通知处理器数据已经准备就绪。现在, 处理器从数据总线上取来数据, 存入存储器的地址单元中。

电源把标准的 220V 交流转换成直流并降低电压, 使之符合计算机的要求。电源的功率通常约为 300W。

处理器

PC 的大脑是处理器(也称为中央处理器或 CPU), 它是建立在 Intel 8086 系列的基础上的, 用来完成所有的指令执行与数据处理。各种处理器的速度、存储器容量、寄存器以及数据总线是不相同的。内部时钟用来同步与控制所有的处理器操作。基本时间单位——时钟周期是按 MHz(每秒钟百万周期)计算的。以下是各种 Intel 处理器的简要介绍:

1. 8088

有 16 位寄存器和 8 位数据总线, 并且最大可以寻址到 1MB 内存。虽然寄存器一次可以处理二个字节, 但数据总线一次只能传送一个字节。该处理器是在**实模式(real mode)**下运行的, 也就是说, 用段寄存器中的实际(“实”)地址, 一次运行一个程序。

2. 8086

类似于 8088, 但有 16 位数据总线, 运行速度较快。

3. 80286

比前两种处理器运行速度都快, 具有附加的能力, 并且最大可寻址到 16MB。这种处理器及其后继者可以在**实模式**或**保护模式(Protected mode)**下工作, 保护模式使操作系统能像 Windows 一样执行多任务(同时运行一个以上的作业), 而且对它们彼此加以保护。

4. 80386

有 32 位寄存器和 32 位的数据总线, 并且最大可寻址达 4GB 的存储器。处理器不但支持

保护模式，而且还支持虚模式(virtual mode)，从而可以把存储器的一部分交换到磁盘上。用这种方法，程序在并发运行时就有了操作空间。

5. 80486

同样有 32 位寄存器和 32 位的数据总线，并增加了快速的高速缓冲存储器，使处理器可以存放最近使用的指令与数据的副本。当直接使用高速缓冲存储器而不必去访问较慢的主存储器时，处理器可以工作得更快。

6. Pentium

有 32 位寄存器和 64 位的数据总线，还有分别用于数据和存储器的各自的高速缓冲存储器(Intel 采用“Pentium”的名字是因为和那些数字编号不同，名字是有版权的)。它的超标量设计使处理器能在每个时钟周期内译码并执行多条指令。

7. Pentium II 和 Pentium III

有双独立总线设计，使到达高速缓存和内存存储器能有各自独立的通路。以前的处理器与在系统板上的高速缓冲存储器连接时会产生延迟，而这些处理器则是以 64 位宽的总线连接到一个内部的高速缓冲存储器上。

80486 之前的处理器都采用通常所说的单级流水线，这就限制它们只有在完成一条指令后才能开始执行下一条指令。流水线所涉及的方法是处理器要把一条指令分成利用不同资源的顺序的一些步骤。Pentium 有 5 级流水线结构，Pentium II 有 12 级超流水线结构。这一特性使它们能并行地运行许多操作。

设计人员所面临的问题是：由于处理器的运行速度要比存储器的动作快得多，所以它不得不等待存储器提供指令。为了解决这个问题，每种先进的处理器就要有更多的动态执行能力。例如，借助于多路的转移预测，处理器先行几步去预测下一步要做什么。

执行部件与总线接口部件

如图 1-2 所示，处理器划分成两个逻辑部件：执行部件(EU)和总线接口部件(BIU)。EU 的作用是执行指令，而 BIU 则是给 EU 提供指令和数据。EU 包含算术逻辑部件(ALU)、控制部件(CU)以及几个寄存器。这些部件用于执行指令和算术与逻辑操作。

BIU 的最重要功能是管理总线控制部件、段寄存器和指令队列。BIU 控制传送数据到 EU、存储器和外部输入/输出设备传送数据的总线，段寄存器则控制存储器寻址。

BIU 另外一个功能是取指令。因为正在执行的程序的指令是在存储器中的，所以 BIU 必须从存储器取出指令，然后把它放入指令队列中，队列规模的不同取决于处理器。这一特性使 BIU 能做到预取指令，以便始终能有一个准备好执行的指令队列。

EU 和 BIU 是并行工作的，同时 BIU 保持领先一步。当 EU 需要从存储器或 I/O 设备取数据时，它会通知 BIU。此外，EU 需要来自指令队列的机器指令。队列顶部的指令是当前可执行的指令，当 EU 正在执行一条指令时，BIU 就从存储器预取下一条指令。这种预取能与执行重叠并提高了处理速度。

程序员不能访问处理器这些部件中的任何一个部件。

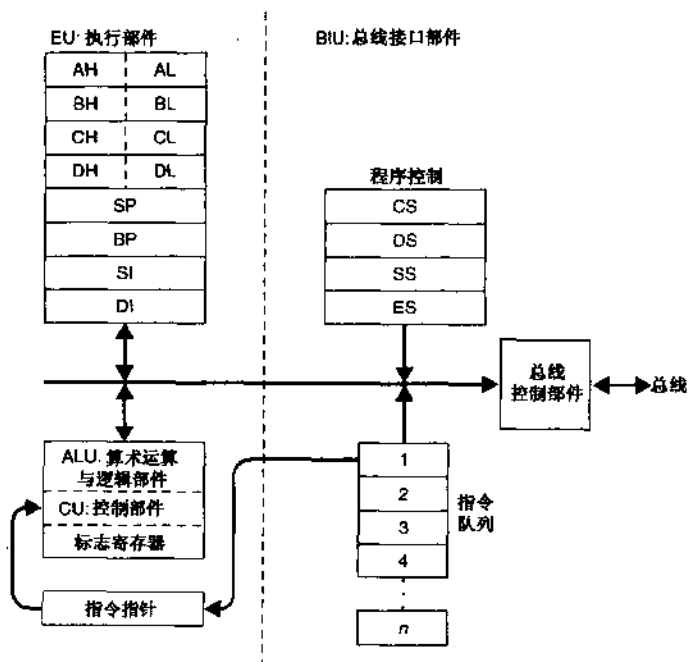


图 1-2 执行部件和总线接口部件

1.7 内存存储器

在 PC 中有两种类型的内存存储器，即随机存取存储器(RAM)和只读存储器(ROM)。存储器中的字节是顺序编号的，从 00 开始使得每个单元都有它唯一编号的地址。

图 1-3 作为一个简单例子所表示的是 8086 类型 PC 的物理存储器映像。在存储器的 1MB 中，前面 640KB 是基本 RAM，它们当中的大多数你是可以使用的。在存储器最底部的中断向量表会在本章稍后加以说明，BIOS 数据区在第 3 章和第 24 章讨论，而视频显示区则要在第 9 章阐述。

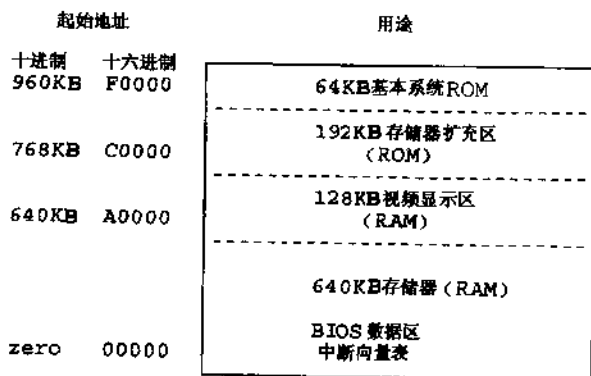


图 1-3 基本存储器映像

ROM

ROM 是一种专门的存储器芯片，顾名思义，它是只读的。因为指令与数据是永久性地“烧进”芯片中的，所以它们是不能修改的。地址从 768KB 开始的 ROM 基本输入/输出系统 (BIOS) 管理输入/输出设备，如硬盘控制器。从 960KB 开始的 ROM 控制计算机的基本功能，如接通电源后的自检，图形用的点模式，以及磁盘的自装入程序。当你接通电源后，ROM 完成各种检查并把专门的系统数据从磁盘装入 RAM。

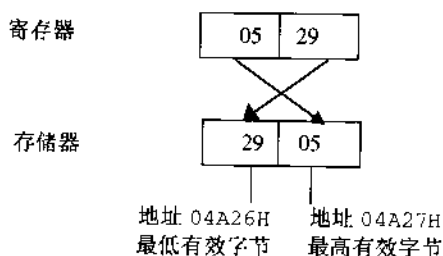
RAM

程序员主要关心的是 RAM，把它叫做“读—写存储器”可能更贴切一些。RAM 可以当作程序暂时存储与执行的“工作单”来使用。当你接通电源以后，ROM 通过引导程序把操作系统的一部分装入 RAM。然后就可以要求它完成动作，如从磁盘上把程序装入到 RAM 中。你的程序在 RAM 中执行并通常会在屏幕、打印机或磁盘上产生输出。当完成以后，你可以要求系统把另一程序装入到 RAM 中，对以前的程序进行覆盖写入。

关闭电源时，会抹掉 RAM 的内容，但对 ROM 没有影响。因此，如果你已经改变了文档中的数据，那么在关闭 PC 之前，你就需要把它先保存在磁盘上。在有关 RAM 的进一步的讨论中，将使用存储器这个通用术语。

通过寻址找到存储器中的数据

根据型号，处理器一次可以存取存储器中一个或多个字节。考虑一个十进制数 1315。这个值的十六进制表示是 0529H，需要存储器的二个字节或一个字。它由高位(最高有效)字节 05 和低位(最低有效)字节 29 组成。处理器在存储器中是按相反字节顺序存放数据的：低位字节在低的存储器地址里，高位字节在高的存储器地址里。例如，处理器把值 0529H 从寄存器传送到存储器地址 04A26H 和 04A27H 处，就像下面这样：



处理器要求存储器中的数值数据按相反字节顺序存放，并且要按这种方法处理数据。当处理器从存储器取回该字节时，再把字节顺序反一下，把它们正确地重新按 hex 0529 存放到寄存器中。虽然这一特性是完全自动的，但在编写和测试汇编语言程序时，你一定要对它加以注意。

当用汇编语言编程时，你必须清楚存储器单元的地址和它的内容并不是一回事。在前面的例子中，地址 04A26H 的内容是 29，而地址 04A27H 的内容是 05。

寻址方案有两种类型：

1. **绝对地址寻址**，比如 04A26H 是一个 20 位的值，它直接对应存储器中的指定单元。
2. **段：偏移地址寻址**，把段的起始地址和偏移地址组合在一起寻址。下一节会详细介绍这一方案。

1.8 段与寻址

段是在程序中定义的一个专门区域，它是一个包括代码、数据以及堆栈的区域。段是从小段边界(paragraph boundary)即能被 16 或 hex 10 除尽的单元开始的。虽然段可以位于存储器的几乎所有地方并在实模式下可以多达 64KB，但它只需要程序所要用的那么多空间，也就是数据与处理这些数据的指令所占用的空间。当一条指令往一个段寄存器中装入段地址时，会自动移掉最右边的 4 个 0。

你可以定义任何数据的段；为了访问一个特定的段，只需要改变适当段寄存器中的地址就可以了。在实模式下，三个主要的段是代码段、数据段和堆栈段：

代码段包括要执行的机器指令。典型地，第一条可执行的指令是在这个段的起点，而操作系统连接到开始执行程序的单元。顾名思义，代码段(CS)寄存器寻址代码段。

数据段包括程序所定义的数据，常量和工作区。数据段(DS)寄存器寻址数据段。

堆栈段包括程序需要暂存的任何数据与地址，或由你自己“调用”的子程序所用的数据与地址。堆栈段(SS)寄存器寻址堆栈段。

1.8.1 段边界

段寄存器的大小是 16 位，包含段的起始地址。图 1-4 说明了有关 SS、DS 和 CS 寄存器以及它们与堆栈段、数据段和代码段关系(寄存器与段不必要按顺序表示)。其他的段寄存器还有 ES(附加段寄存器)，80386 和更新的处理器中还有 FS 与 GS 寄存器，它们为存储数据提供了附加段。

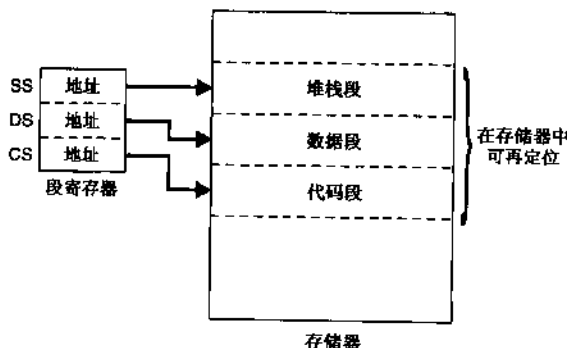


图 1-4 段与寄存器

如前所述，一个段起始于小段边界，它是一个可以被十进制 16 或 hex 10 除尽的地址。假定一个数据段的起点是存储器单元 038E0H，由于在这种及所有其他情况下，最右边十六进制数位都是 0，所以存放数字 0 到段寄存器中是没有必要的。因此，038E0H 是以 038E 存

放在寄存器中的，最右边的 4 位被移掉了。在适当的场合，本书使用方括号指明最右边的十六进制的 0，如 038E[0]。

1.8.2 段偏移值

在一个程序中，一段范围内的所有存储单元都是相对于段起始地址的。我们把一个段内从段地址到另一个单元以字节为单位的距离称为**偏移值**(或位移量)。在实模式下，2 字节(16 位)的偏移值的范围可以从 0000H 到 FFFFH，或是从 0 到 65535。因此，代码段的第一个字节在偏移值 00 处，第二个字节在偏移值 01 处，依此类推，直到偏移值 65535。为了访问一个段内的存储器任何存储单元，处理器要把在段寄存器中的段地址与该单元的偏移值(即它与段起始点的字节距离)组合起来。

假设数据段的起点是在 038E0H 单元，DS 寄存器包含数据段的段地址 038E[0]H，而指令访问的单元距数据段起点的偏移值是 0032H 字节。为了访问所要求的单元，处理器就要把数据段的地址和偏移值组合在一起：

DS 段地址:	038E0H
偏移值:	+ 0032H
实际地址:	03912H

因此，由指令访问的字节的实际存储单元是 03912H。注意：程序包含一个或多个段，它们可以从存储器的任何地方开始，大小也可以不同，并且可以排成任何序列。你仅仅会偶然涉及到段地址和偏移值，这是因为汇编程序和处理器会处理它们。但是，你还需要知道它们的含义和用法。

1.9 寄 存 器

处理器的寄存器用于控制指令执行，存储器寻址，并提供算术运算能力。程序按名字访问寄存器，如 CS，DS，以及 SS。在寄存器中的位(如同在字节中的位一样)一般是从 0 开始从右到左编号，如：

... 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1.9.1 段寄存器

段寄存器为一个称为当前段的区提供寻址。PC 系列所用的 Intel 处理器提供不同的寻址能力。

1. 8086/8088 寻址

这些处理器的段寄存器长度是 16 位，工作在实模式下。因为段地址是在小段边界上的(可被 16 或 hex10 除尽的)，它的地址最右边 4 位是 0。如前所述，段地址是存放在段寄存器中的，而且处理器移掉了最右边的 4 个 0 位，所以十六进制的 nnnn0 就变成了 nnnn。例如，十六进制地址 038E0 是按 038E 存放在段寄存器中的。处理器承担了最右边的 4 位，所以段寄

寄存器的有效长度是 20 位。

FFFF[0]H 的允许寻址范围最大值达到 1048560 个字节。如果你不能确定，那么每个 hex F 译码成二进制的 1111，考虑到最右边的 4 个 0 位，所以再加上 4 位为 1 的值。

2. 80286 寻址

在实模式下，80286 处理器处理寻址和 8086 是一样的。在保护模式下，24 位的寻址方案提供的寻址范围达到 FFFFF[0]，即 16MB。段寄存器作为选择器，用来从存储器中存取 24 位段地址。保护模式允许处理器实现多任务，这时它能从正在执行的任务(或程序)转换到另一任务。系统必须能够保护存储器中的每个任务不受其他任务的影响。

3. 80386/486/Pentium 寻址

在实模式下，这些处理器也能处理像 8086 那样多的寻址。在保护模式下，处理器用 48 位寻址，这时允许寻址的段可达 4MB。16 位的段寄存器作为选择器用来从存储器中存取 32 位段地址。

6 个段寄存器是 CS、DS、SS、ES、FS 以及 GS。

1. CS 寄存器

有效程序代码段的起始地址。这个段地址加上在指令指针(IP)寄存器中的偏移值，就指明了为执行指令所要取得的指令地址。在通常的编程中，不需要直接访问这个寄存器。

2. DS 寄存器

包含程序数据段的起始地址。指令使用这一地址放置数据；该地址加上在指令中的偏移值，就可以访问位于数据段中的指定字节单元。

3. SS 寄存器

容许在存储器中实现堆栈，程序可用来暂时存放地址和数据。系统在 SS 寄存器中存放程序堆栈段的起始地址。这一段地址加上堆栈指针(SP)寄存器中的偏移值，就指明了正被寻址的堆栈中的当前字。在通常的编程中，不需要直接访问 SS 寄存器。

4. ES 寄存器

在某些串(字符数据)操作中，用于存储器寻址。在这方面，ES(附加段)寄存器是和 DI(变址)寄存器相关联的。如果程序需要使用 ES，你就必须用一个适当的段地址将它初始化。

5. FS 和 GS 寄存器

是 80386 为处理存储器请求而采用的备用的附加段寄存器。

1.9.2 指针寄存器

指针寄存器有 32 位的 EIP、ESP 和 EBP；最右边 16 位的部分分别是 IP、SP 和 BP。

1. 指令指针(IP)寄存器

16 位的 IP 寄存器包含要执行的下一条指令的偏移地址。IP 是和 CS 寄存器相关联的(如 CS:IP), 其中 IP 指明现在正在执行的代码段内的当前指令。通常是不能在程序中访问 IP 的, 但你可以改变它的值, 例如: 在使用 DEBUG 程序去测试一个程序的时候。80386 采用了一个扩充的 32 位 IP, 叫做 EIP。

在以下的例子中, CS 含有 39B4[0]H, IP 含有 514H。为了找到要执行的下一条指令, 处理器要把在 CS 中的地址与在 IP 中的偏移值组合在一起:

在 CS 中的段地址	39B40H
加上在 IP 中的偏移值	+ 514H
下一条指令的地址	3A054H

对于每条执行的指令, 处理器都要改变在 IP 中的偏移值, 以便 IP 能有效地指向每个执行步骤。

SP(堆栈指针)和 BP(基址指针)寄存器是与 SS 寄存器相关联的, 并且容许系统访问在堆栈段中的数据。处理器自动管理这些寄存器。

2. 堆栈指针(SP)寄存器

16 位的 SP 寄存器提供一个偏移值, 当它和 SS 寄存器相关联时(SS:SP), 它可以访问在堆栈中正在处理的当前字。80386 采用了一个扩充的 32 位堆栈指针, 即 ESP 寄存器。

在下面的例子中, SS 寄存器含有段地址 4BB3[0]H, SP 含有偏移值 412H, 为了找到在堆栈中正在处理的当前字, 处理器要把 SS 中的地址与 SP 中的偏移值加以组合:

在 SS 中的段地址	4BB30H
加上在 SP 中的偏移值	+ 412H
在堆栈中的地址:	4BF42H

3. 基址指针(BP)寄存器

16 位的 BP 简化了对参数的访问, 这些参数是程序通过堆栈传递的数据和地址。处理器把在 SS 中的地址和在 BP 中的偏移值组合起来, BP 还可以和 DI 或 SI 组合起来, 作为用于特殊寻址的基址寄存器。80386 采用了扩充的 32 位 BP, 即 EBP 寄存器。

1.9.3 通用寄存器

通用寄存器是 32 位的 EAX, EBX, ECX, 以及 EDX; 最右边的 16 位部分分别是 AX, BX, CX 以及 DX。例如, AX 是 EAX 最右边的 16 位。AX 本身由二部分组成: 最左边 8 位(“高”的部分)称为 AH; 最右边的 8 位(“低”的部分)称为 AL。这一特性允许你去访问任何部分——EAX, AX, AH, 或 AL——按照名称分别处理双字, 字或字节。

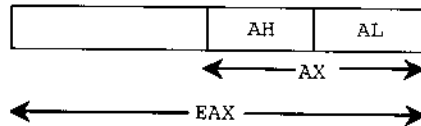
以下的汇编语言指令说明把 0 分别传送到 AX, BH 和 ECX 寄存器:

```
MOV AX, 00
MOV BH, 00
```

```
MOV ECX, 00
```

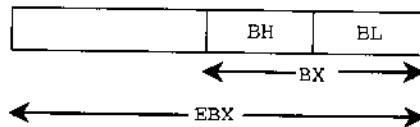
1. AX 寄存器

AX——主累加器，用于有关输入/输出和大多数算术运算操作。例如：乘法，除法及换码指令都使用 AX。此外，某些指令如果访问 AX 而不是访问其他寄存器就会生成更为有效的机器代码。8 位的 AH 和 AL 分别是 16 位 AX 的左边和右边的部分，而 AX 则是 32 位的 EAX 的最右边的 16 位：



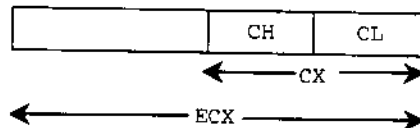
2. BX 寄存器

BX 即基址寄存器，它只是个通用寄存器，可以用作扩展寻址的变址。BX 另一个用途是做计算。BX 还可以和 DI 或 SI 组合起来作为专用于寻址的基址寄存器。8 位的 BH 和 BL 分别是 16 位的 BX 的左边与右边两部分，而 BX 则是 32 位的 EBX 最右边的 16 位：



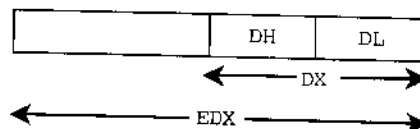
3. CX 寄存器

CX 即计数寄存器。它可以包含控制循环的重复次数的值或左、右移位的次数。还可以用 CX 进行许多计算。8 位的 CH 和 CL 分别是 16 位的 CX 左边与右边的两部分，而 CX 则是 32 位的 ECX 最右边 16 位：



4. DX 寄存器

DX 即数据寄存器。一些输入/输出操作需要使用它，涉及很大值的乘法与除法操作是把 DX 和 AX 配成对合在一起使用。8 位的 DH 和 DL 分别是 16 位的 DX 的左边与右边两部分，而 DX 则是 32 位的 EDX 的最右边 16 位：



通用寄存器可以用作 8 位、16 位或 32 位的加法与减法：

```
MOV  EAX, 225    ; 把 225 传送到 EAX (双字)
ADD  AX, CX       ; CX 加到 AX (字)
SUB  BL, AL       ; BL 减去 AL (字节)
```

1.9.4 变址寄存器

变址寄存器是 32 位的 ESI 和 EDI，它们最右边的 16 位部分分别是 SI 和 DI。这些寄存器可用于变址寻址，并且也可用于某些加法与减法中。

1. SI 寄存器

16 位的源变址寄存器是某些串(字符)处理操作所需要的。在本书中，SI 是和 DS 寄存器联合在一起的。80386 采用了 32 位扩充寄存器 ESI。

2. DI 寄存器

16 位的目的变址寄存器也是某些串操作所需要的。在本书中，DI 是和 ES 寄存器联合在一起的。80386 采用了 32 位扩充寄存器 EDI。

1.9.5 标志寄存器。

32 位标志寄存器所包含的位用来指明不同的活动状态。这种标志寄存器最右边的 16 位部分是标志寄存器，这 16 位中的 9 位是指明计算机的当前状态和处理结果的。许多含有比较与算术运算的指令会改变标志的状态，这样，某些指令就可以通过测试某一位来确定下一步的动作。

以下简要地讨论一下这些公用的标志位：

OF(溢出) 指出在算术运算后高阶(最左边)位的溢出。

DF(方向) 确定传送或比较串(字符)数据时的左右方向。

IF(中断) 指明所有外部中断(如键盘输入)是处理还是不去管它。

TF(陷阱) 允许处理器以单步方式工作。调试程序(如 DEBUG)要设置陷阱标志，便于一步步地通过每次执行单条指令的办法，去检查它对寄存器和存储器的影响。

SF(符号) 包含一次算术运算操作的结果的符号(0=正，1=负)。

ZF(零) 指明算术运算或比较操作的结果(0 等于结果为非零，1 等于结果为零)。

AF(辅助进位) 包含在算术运算操作中第 3 位到第 4 位的进位输出，用于专门的算术运算。

PF(奇偶) 指明在操作产生的结果中为 1 的位的数量。是偶数，则奇偶性为偶；是奇数，则奇偶性为奇。

CF(进位) 包含一个算术运算操作后，来自高阶(最左边)位的进位；还包含移位或循环操作最后位的内容。

标志寄存器中的标志位的位置如下(你不必记住它们)：

标志					O	D	I	T	S	Z		A		P		C
位号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

对于比较和算术运算操作而言与汇编程序最有关系的标志，是 OF，SF，ZF，以及 CF。DF 用于串操作的定向。一些其他的标志作为内部使用，主要和保护模式有关。第 7 章会对标志寄存器作更详细的说明。

1.10 硬件中断

某些事件会使处理器挂起当前的操作并为引起中断的原因去做一些事情。通常，事件是正常的和预期的，比如来自键盘的输入请求。处理器中断当前的操作，调用 BIOS 的例行程序去处理键盘请求，然后返回到被中断的程序。另外一些中断可能是危险的，比如企图除以零，这会造成系统终止处理过程。

另一类中断是软件中断，由程序本身发出一个请求，比如要在屏幕上显示数据。系统要再次进行控制，处理中断，然后返回到被中断的程序。

1.11 要 点

- 处理器只能识别一个位是 0(“关闭”)还是 1(“开通”)，并只能完成二进制算术运算。
- 二进制数的值是由它的位位置决定的。例如，二进制值 1101 等于(从右到左) $2^0+0^1+2^2+2^3$ ，即 13。
- 负的二进制数是用二进制补码来表示的：它的原码表达式按位求反，然后加 1。
- 一个存储器单元是一个字节，由 8 个数据位和 1 个奇偶位组成。2 个相邻的字节组成一个字，4 个相邻的字节组成一个双字。
- 十六进制格式是一种速记的记数法，用来表示 4 位的组。十六进制数字 0~9 和 A~F 代表二进制值 0000 到 1111。
- 字符格式采用 ASCII 码表示数据。
- PC 的核心——处理器按相反字节顺序，以字和双字的形式在存储器中存放数值数据。
- 内存储器的两种类型是 ROM 和 RAM。
- 汇编语言程序由一个或多个段组成：堆栈段用于保存返回地址，数据段用于定义数据和工作区，而代码段则是用于可执行的指令。在段范围内的单元被表示成相对于段起始地址的偏移值。
- CS，DS 和 SS 寄存器分别用来对代码段、数据段和堆栈段寻址。
- CS：IP 的含义是段：下一条要执行的指令的偏移地址。
- SP 和 BP 指针寄存器是与 SS 寄存器联合在一起的，并且允许系统访问堆栈中的数据。
- AX、BX、CX 以及 DX 是系统承担繁重任务的通用寄存器。最左边的字节是“高”位部分，而最右边的字节是“低”位部分。AX(主累加器)用于输入/输出和大多数算术运算。BX 还可以用作扩展寻址的变址。CX 用于计算与专用计数，DX 也可以用于

计算。EAX、EBX、ECX 以及 EDX 是 32 位的通用寄存器。

- SI 和 DI 变址寄存器可以用于扩展寻址，也可以用于加法与减法。这些寄存器还用于某些串(字符)操作。
- 标志寄存器指明处理器的当前状态和执行指令的结果。

1.12 习 题

1-1. (a)计算机存储的基本构件是什么?(b)它的两个状态是什么?

1-2. (a)在 1-1 题中提到的 9 个元素的集合是什么?(b)8 个元素是做什么用的?(c)第 9 个元素的用途是什么?

1-3. 给出以下数据项的长度：(a)小段，(b)字，(c)双字，(d)字节，(e)千字节。

1-4. 将以下十进制数转换成二进制格式：(a)7，(b)15，(c)25，(d)28，(e)33。

1-5. 将以下 8 位二进制数相加：

(a) 00101101	(b) 00110110	(c) 00101111	(d) 01111111
00000101	00111001	00000001	01010101

1-6. 把 1-5 题中的每个二进制数及它们的和转换成十进制值并检查所得的和是否正确。

1-7. 给出以下二进制数的二进制补码：(a)00110110，(b)00111101，(c)01111100，(d)00000000。

1-8. 把以下负的二进制数转换成正的二进制值：(a)11001100，(b)10110111，(c)10101010，(d)11111111。

1-9. (a)将以下 2 个 8 位二进制数相加：11001011 和 01110111。(b)8 位和的十进制值是多少？(c)把二进制值转换成十进制值并检查和是否正确。

1-10. 给出以下值的十六进制表示：(a)ASCII 字母 R，(b)ASCII 数 7，(c)二进制数 01110101，(d)二进制数 01110110。

1-11. 把以下十六进制数相加：

(a) 13B4	(b) 53CD	(c) 8798	(d) DCBE	(e) FDAC
+0033	+0004	+0777	+35B5	+0BAF

1-12. 确定以下十进制数的十六进制表示。转换方法参考附录 A。并把十六进制值转换为二进制值后把 1 的位相加，检查结果正确性。(a)23，(b)37，(c)75，(d)255，(e)4,095，(f)56217。

1-13. 参照附录 B，给出以下 ASCII 字符的位配置：(a)G，(b)S，(c)#，(d)6，(e)*，(f)冒号(：)。

1-14. 处理器的主要功能是什么？

1-15. 区别 PC 存储器的两种主要类型并指出它们的主要用途。

1-16. 说明处理器在存储器中是如何存放以下十六进制值的：(a)2AB5，(b)0AD4C2。

1-17. 解释以下术语：(a)段，(b)偏移值，(c)地址边界。

1-18. (a)段的三种类型是什么? (b)它们的最大范围是多少? (c)它们的起始地址是什么?

1-19. CS, DS, ES 以及 SS 段寄存器的用途各是什么?

1-20. (a)SS 的内容是 2AB4[0]H 和 SP 的内容是 24H, (b)CS 的内容是 2BC3[0]H 和 IP 的内容是 3AH, 求它们所形成的绝对地址。

1-21. 说明用于以下目的的是哪些寄存器: (a)循环计数, (b)乘法与除法, (c)对段寻址, (d)指明零结果, (e)一条将要执行指令的偏移地址, (f)加法与减法。

1-22. 表示出 EBX 寄存器以及 BH、BL 和 BX 在其中的位置并说明它们的大小。

1-23. 用汇编语言指令编码, 将传送(MOV)值 50 到以下每个寄存器中: (a)CX, (b)CH, (c)CL, (d)ECX。

1-24. 用指令编码, 把 28 的值加(ADD)到以下寄存器中: (a)CL, (b)CH, (c)CX, (d)ECX。

1-25. 指出受以下动作影响的标志: (a)算术运算的和是零, (b)算术运算的和是负的, (c)串数据由左向右传送。

目的：说明在 PC 上装入与执行程序的一般要求。

2.1 引言

在这一章里，我们讨论 PC 的软件环境：操作系统的功能和其主要组成部分。我们要探讨引导过程(当你的计算机加电时，系统本身是如何装入的)，要研究系统是如何装入将要执行的程序，系统怎样使用堆栈，以及在代码段中一条指令是如何引用数据段中的数据。

本章完成对 PC 硬件和软件的基本阐述，并且使你能进入到第 3 章，在那一章里，你可以把简单的程序键入到存储器中并一步步地执行它们。

2.2 操作系统的特点

操作系统提供通用地、设备无关地访问计算机资源(如键盘、屏幕和磁盘驱动器这样的一些设备)的能力。设备无关的意思是，你不必专门地去访问设备，因为系统可以在设备级处理输入/输出(I/O)操作，与要求这一操作的程序无关。

与我们有关的一些功能如下：

- **文件管理。**操作系统维护在系统磁盘上的目录与文件。程序负责建立与修改文件，而系统则负责管理它们在磁盘上的定位。
- **输入/输出。**程序需要借助中断从系统输入数据或向系统输出数据。程序员可以减轻在 I/O 低层次上的编码负担。
- **程序装入。**当用户或程序要求执行程序时，程序的装入程序会处理从磁盘上取出程序，读入存储器中，以及为了执行程序而进行的初始化等这样一些步骤。
- **存储器管理。**当程序的装入程序为执行一个程序而把它从磁盘装入到存储器时，它会为程序代码和数据在存储器中分配足够大的空间。程序可以在它的存储区里处理数据，可以释放不需要的存储区，而且还可以要求追加存储区。
- **中断处理。**系统可借助于中断访问外部设备。

2.3 BIOS 引导过程

接通计算机电源的时候，会使处理器进入复位状态，把所有存储单元清除为 0，完成存储器的奇偶校验，以及把 CS 寄存器设置成段地址 FFFF[0]H，并把 IP 寄存器设置成偏移值为 0。因此，第一条要执行的指令是在由 CS:IP 对所形成的地址里的，这个地址是 FFFF0H，是 ROM 中的 BIOS 入口点。

BIOS 包含在 ROM 中的一组例行程序提供对设备的支持。起始单元为 FFFF0H 的 BIOS 例行程序检查各种端口去识别与初始化一些设备，这些设备被连接到计算机上并提供在设备上读写各种服务。因此，BIOS 要建立两个数据区：

1. **中断向量表**。它开始于低端存储器的 0 单元并包含 256 个 4 字节地址，其格式是：段：偏移值。BIOS 和操作系统在中断发生时使用这些地址。
2. **BIOS 数据区**。它开始于 40[0]单元，大部分与所连接设备的状态有关。我们将在第 24 章详细讨论。

下一步 BIOS 要确定磁盘是否包含现存的系统文件，如果有，它就要访问来自磁盘的引导装入程序。由这个程序把系统文件从磁盘装入内存，并把控制权交给它们。系统文件包括设备驱动程序和其他硬件专用代码。这些模块初始化内部系统表和中断向量表的系统部分。

操作系统的一项任务是当需要访问 BIOS 的设备时能和 BIOS 连接。当用户程序请求 I/O 操作时，它要传送请求给 BIOS，在 BIOS 的控制下访问所需要的设备。但是，有时程序会直接向 BIOS 发出请求，如对于键盘和屏幕的服务。在其他时间——虽然很难得而且不提倡——程序可以绕过操作系统与 BIOS 直接访问设备。图 2-1 表明这些可供选择的通路。

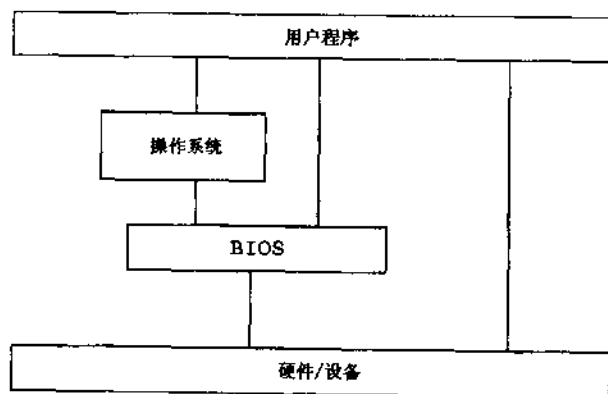


图 2-1 输入/输出接口

2.4 系统程序的装入程序

一旦 BIOS 把控制交还操作系统，你就可以请求执行程序。可执行程序有两种类型：.COM

和.EXE。 .COM 程序由一个包含代码、数据和堆栈的段组成。 .COM 程序作为小的实用程序或常驻程序(安装在存储器内,在其他程序运行期间它是可用的)是有用的。在实模式下, .EXE 程序是由各自独立的代码段、数据段和堆栈段组成,是一种较为正式的程序。本书采用这两种类型的程序。

要求系统把.EXE 程序从磁盘装入存储器时,装入程序是按以下步骤执行的:

1. 从磁盘上取.EXE 程序。
2. 在可用内存存储器的一个小段边界上,构造一个 256 字节(100H)的程序段前缀(PSP)。
3. 紧随 PSP 的下一地址。把程序装入存储器中
4. 把 PSP 的地址装入 DS 与 ES 寄存器。
5. 把代码段地址装入 CS 寄存器并把 IP 寄存器设置成代码段中第一条指令的偏移值(通常是 0)。
6. 把堆栈段地址装入 SS 寄存器并把 SP 寄存器设置成堆栈的大小。
7. 传送控制给要执行的程序,通常从代码段的第一条指令开始。

通过上述方法,程序的装入程序将 CS:IP 和 SS:SP 正确地初始化。但是请注意,程序的装入程序在 DS 和 ES 中都存放了 PSP 的地址,而在这些寄存器中你的程序需要数据段的地址。因此, .EXE 程序必须按数据段地址去初始化 DS,正如你在第 4 章将会看到的那样。

现在,我们来研究堆栈段、代码段与数据段。

2.5 堆 栈

.COM 程序与.EXE 程序都需要在程序中保留一个区作为堆栈。堆栈有 3 个主要的用途:

1. 当调用一个子程序时,程序要在堆栈中保留返回地址,用于子程序返回。
2. 调用子程序的程序还可以用把数据放在堆栈中的办法来传送数据,子程序可以通过堆栈存取这些数据。
3. 当程序要使用寄存器计算时,可以在堆栈中保存寄存器当前的内容,进行计算后,再把数据从堆栈恢复到寄存器中。

装入程序自动地为.COM 程序定义堆栈,而必须为.EXE 程序明确地定义堆栈。在实模式下,堆栈中的每个数据项是一个字(2 个字节)。SS 寄存器在被装入程序初始化时,包含了堆栈的起始地址。最初,SP 寄存器包含堆栈的大小,它的值所指向的字节超过堆栈的末端。堆栈在存储数据的方法上是和其他段不同的:它从段的最高的单元开始存放数据,而且存放数据的顺序是向下(或相反方向)通过存储器的。

PUSH 和 POP 是那些修改 SP 寄存器内容的若干指令中的两条,用来在堆栈中存放数据和取回数据。PUSH 执行 SP 减 2,指向堆栈中下一个较低的存储字并存放一个值在那里(或称“进栈”);POP 执行从堆栈回送一个值并使 SP 加 2,去指向下一个较高的存储字(或称“出栈”)。

对于一个特定的过程,将全部数据进栈,保存在堆栈的一个部分,这进栈部分称为堆栈帧。下面的例子说明把 AX 与 BX 寄存器的内容进栈,然后顺序地从堆栈中将数据出栈再返回到寄存器。假定 AX 的内容是 hex 026B, BX 的内容是 04E3, SP 内容是 36(在 SS 中的堆

栈的段地址在这里与我们无关)。

1. 最初, 堆栈是空的并如下所示:

偏移值	堆栈帧	SP=36
34	0000	
32	0000	
30	0000	
3E	0000	

2. PUSH AX: SP 减 2(到 34), 并把 AX 的内容 026B 存入堆栈。注意, 操作使所存放的字节次序变反了, 即 026B 变成了 6B02:

偏移值	堆栈帧	
34	6B02	← SP=34
32	0000	
30	0000	
2E	0000	

3. PUSH BX: SP 减 2(到 32)并把 BX 的内容 04E3 存入堆栈, 成为 E304:

偏移值	堆栈帧	
34	6B02	
32	E304	← SP=32
30	0000	
2E	0000	

4. POP BX: 把堆栈中 SP 所指向的字(E304)恢复到 BX 中, 并且 SP 加 2(到 34)。由于字节是正确恢复的, 所以 BX 现在的内容是 04E3:

偏移值	堆栈帧	
34	6B02	← SP=34
32	E304	
30	0000	
2E	0000	

5. POP AX: 把堆栈中 SP 所指向的字(6B02)恢复到 AX 中, 并且 SP 加 2(到 36)。由于字节是正确恢复的, 所以 AX 现在的内容是 026B:

偏移值	堆栈帧	SP=36
34	6B02	
32	E304	
30	0000	
2E	0000	

注意: POP 指令是以和 PUSH 指令相反的顺序编码的, 在这个例子中按照这样的顺序进

栈时是 AX 和 BX，而出栈时是 BX 和 AX。而且进入堆栈的值仍然在那里，尽管 SP 指针已经不再指向它们。随后的 PUSH 操作将会用新的值取代旧的值。

要确保你的程序进入堆栈的值和堆栈出栈的值是等同的。尽管这是一个相当简单的要求，但是如果使用不当就会导致严重的程序错误。另外，对于一个 .EXE 程序，堆栈必须足够大，以便它能装下可能进栈的所有值。注意，当 SP=0 时，堆栈是满的。

其他的一些与进栈或出栈有关的指令是：

- PUSHF 和 POPF：保存和恢复标志的状态。
- PUSHA 和 POPA(80286+)：保存和恢复所有通用寄存器(AX、CX、DX、BX、SP、BP、SI 以及 DI)的内容并使 SP 加/减 16。
- PUSHAD 和 POPAD(80386+)：保存和恢复所有扩充寄存器(EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI)的内容并使 SP 加/减 32。

2.6 指令的执行与寻址

汇编语言程序员用符号代码编写程序并用汇编程序把它翻译成机器码，如 .COM 或 .EXE 程序。为了执行程序，系统只把机器码装入到存储器中。每条指令至少要包含一种操作，如传送、加或返回。依据不同的操作，一条指令可以有-一个或多个操作数，用来访问该操作要处理的数据。

在执行指令的过程中，处理器所采取的基本步骤是：

1. 从存储器中取出要执行的下一条指令并把它放在指令队列中。
2. 指令译码：计算访问存储器的地址，把数据传送到算术逻辑部件中，并且增加指令指针(IP)寄存器的值。
3. 执行指令：完成所要求的操作，把结果存入寄存器或存储器中，并且按要求设置标志位(如零或进位)。

Pentium 的流水线功能使它能重叠进行操作，如以下三条指令的简单举例：

1	取指	译码	执行	
2		取指	译码	执行
3			取指	译码
			执行	

正如已经讨论过的，对于 .EXE 程序，CS 寄存器提供程序代码段的起始地址，DS 提供数据段的起始地址。代码段包含要被执行的指令，数据段则包含指令要访问的数据。IP 寄存器指明在代码段中当前指令的偏移地址，指令操作数指明在数据段中要被访问的偏移地址。

考虑一个例子，在这个例子中，程序的装入程序已确定把一个 .EXE 程序装入到起始地址为 05BE0H 的存储区中。装入程序相应地用段地址 05BE[0]H 初始化 CS，并且用 0 初始化 IP。CS：IP 一起确定了要执行的第一条指令的地址：5BE0H+0000H=5BE0H。用这种方法，使代码段中的第一条指令开始执行。如果第一条指令是 2 字节长，则处理器使 IP 加 2，这样下一条要执行的指令就在 5BE0H+2=5BE2H 单元中。

假定程序继续执行，并且 IP 当前的内容是偏移值 0023H。现在 CS：IP 确定下一条要执行指令的地址如下：

CS 段地址：	5BE0H
IP 偏移值：	+ 0023H
指令地址：	5C03H

假设起始于 05C03H 的 MOV 指令要把在存储器中的一个字节内容复制到 AL 寄存器中，该字节在数据段中的偏移值为 0016H。下面是这一操作的机器码与符号码：

A01600 MOV AL, [0016]

地址 05C03H

地址 05C03H 包含的是处理器要访问的机器码指令的第一个字节(A0)。第二与第三个字节包含的是以相反字节顺序存放的偏移值(0016 是按 1600 存放的)。在符号码中，在方括号(变址操作符)中的操作数[0016]指明一个偏移值，它和实际存储器地址 16 是有区别的。

假设程序是用数据段地址 05D1[0]H 来初始化 DS 寄存器的。为访问数据项，处理器是由 DS 中的段地址加上指令操作数中的偏移值(0016H)来确定它的存储单元。因为 DS 的内容是 05D1[0]H，所以所访问的数据项的实际单元是：

DS 段地址：	5D1CH
段偏移值：	+ 0016H
数据项地址：	5D26H

假设地址 05D26H 含有 4AH，处理器现在要在地址 05D26H 上提取 4AH 并把它复制到 AL 寄存器。如图 2-2 所示，CS 指向代码段的起始点，DS 指向数据段的起始点，在代码与数据段中的偏移值，以及 4A 的值被复制到 AL。

随着处理器取指令中的每个字节，IP 寄存器加 1。因为 IP 原来的内容是 23H，并且执行的机器码是 3 个字节，所以 IP 现在的内容是 0026H，它是下一条指令的偏移值。处理器现在准备好执行这条指令，它再一次由 CS 中的段地址(05BE0H)加上 IP 中当前的偏移值(0026H)来确定存储地址，即 05C06H。

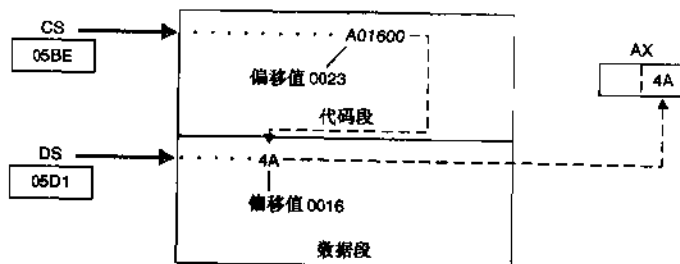


图 2-2 段与偏移值

指令也可以访问一个以上的字节。例如，假定一条指令要把 AX 寄存器的内容(0248H)存入数据段中起始于偏移值 0016H 的 2 个相邻字节中。该指令的符号码是：MOV [0016], AX。处理器是以相反字节顺序把 2 个字节存放在存储器中的，如：

字节内容:	48	02
数据段中的偏移值:	0016	0017

另一条指令 `MOV AX, [0016]` 随后可以使用把这些字节从存储器复制到 `AX` 的办法, 重新取回它们。这一操作把字节顺序变反并正确地存放在 `AX` 中, 成为 `0248`。

2.7 指令的操作数

指令可以没有操作数, 也可以有 1、2 或 3 个操作数。使用标准的名字和方括号中的名字与数字, 可以清晰地表示操作数。在以下的例子中, `DW` 定义 `WORDX` 为一个字(两个字节):

```
WORDX DW 0           ; 把 WORDX 定义为字
...
MOV CX, WORDX        ; 把 WORDX 的内容传送到 CX
MOV CX, 25           ; 把值 25 传送到 CX
MOV CX, BX           ; 把 BX 的内容传送到 CX
MOV CX, [BX]         ; 把 BX 中的地址单元的内容传送到 CX
```

- 第一个 `MOV` 在存储器(`WORDX`)与寄存器(`CX`)之间传送数据。
- 第二个 `MOV` 传送立即数(25)到寄存器(`CX`)。
- 第三个 `MOV` 在寄存器之间传送数据(`BX` 到 `CX`)。
- 第四个 `MOV` 中的方括号定义了一个变址操作符, 其含义是: 利用在 `BX` 中的偏移地址(与 `DS` 中的段地址组合起来, 成为 `DS:BX`)去定位存储器中的一个字, 并把它的内容传送到 `CX`。比较这条指令和第三条 `MOV`(简单地把 `BX` 的内容传送到 `CX`)的作用。第 6 章将详细说明这种间接寻址。

2.8 保护模式

在 Windows 保护模式下, 处理器可以从一个任务切换到另一个任务。每个程序都有自己的存储区, 而且处理器保护每个区域并保存每个程序的状态。在实模式下, 段寄存器包含实际的段地址, 但只限于 1MB 的寻址。然而, 保护模式需要更大的寻址能力。

为了这一目的, 它使用了各种表格, 这些表格包括:

- 局部描述符表(local descriptor table)。取决于系统, 每个任务有一个表。16 位的 `LDT` 寄存器包含正在执行的当前任务的这张表的地址。
- 中断描述符表(interrupt descriptor table)。此表处理中断操作, `IDT` 寄存器包含其地址。
- 全局描述符表(global descriptor table)。此表包含每个局部描述符表的地址, 32 位的 `GDT` 寄存器包含这个表的地址。

段的实际地址是存放在描述符表中的, 段寄存器(或称选择器)存放的是指向当前局部描

述符表的指针。该表提供 32 位寻址，可达 4.3GB。

通常，中断向量表是放在 0000:0000 单元，并且每个任务可以有这个表的副本。

2.9 要 点

- 接通计算机电源使处理器进入复位状态，把全部存储器单元清除为 0，完成存储器的奇偶校验，并且把 CS 和 IP 寄存器设置成在 ROM 中的 BIOS 的入口点。
- 两种可执行程序类型是.COM 和.EXE。
- 为了装入.EXE 程序，装入程序要在存储器的小段边界上建立一个 256 字节(100H)的 PSP 并紧跟 PSP 立即存放此程序。然后在 DS 与 ES 寄存器中装入 PSP 的地址，把代码段的地址装入 CS，设置 IP 为代码段第一条指令的偏移值，把堆栈的地址装入 SS，并把 SP 设置成堆栈的大小。最后，装入程序传送控制给要执行的程序。
- 堆栈的用途是提供地址和数据项的暂存空间。
- 程序的装入程序为.COM 程序定义堆栈，而你必须明确地为.EXE 程序定义堆栈。
- 当处理器取指令的每个字节的时候，它要使 IP 寄存器增量，使得 CS:IP 对包含要执行的下一条指令的段：偏移值地址。

2.10 习 题

2-1. 指出操作系统的 5 种主要功能。

2-2. 给出在引导过程中系统所采取的步骤。

2-3. 程序的装入程序在它装入要执行的模块时，在可执行模块的前面要建立和存放一个数据区。(a)这个数据区叫什么名字？(b)它的大小是多少？

2-4. 程序的装入程序在它装入一个有待执行的.EXE 程序时，要完成某些操作。装入程序用什么样的值来初始化下列寄存器：(a)SS 和 SP，(b)CS 和 IP，(c)DS 和 ES。

2-5. 说明堆栈的用途。

2-6. 说明对于(a).COM 程序和(b).EXE 程序，堆栈是如何定义的(即谁来定义堆栈或堆栈定义成什么样的)。

2-7. (a)栈顶最初在什么地方？如何确定它的地址的？(b)堆栈中每项的大小是多少？

2-8. 给定一堆栈定义为 DW 32H，当(a)堆栈是空的和(b)堆栈是满的时，SP 的内容是什么？

2-9. 堆栈的大小被定义为 64H 个字节。(a)SP 寄存器中的初始值是什么？(b)CX 内容为 25A4H，在 PUSH CX 之后，表示出堆栈和 SP 的内容。(c)DX 的内容是 3B2AH，在 PUSH DX 之后，表示出堆栈和 SP 的内容。(d)指出在 POP DX 和 POP CX 之后 SP 的内容。

2-10. 在程序执行期间，CS 的内容为 4AB6[0]，SS 的内容为 4A82[0]，IP 的内容为 36H，以及 SP 的内容为 28H(这些值是按正常的而不是按反向字节顺序表示的)。计算(a)栈顶(当前单元)的地址和(b)要执行的指令的地址。

2-11. 在程序执行期间, CS 的内容为 5C9B[0], SS 的内容为 5C84[0], IP 的内容为 48H, 以及 SP 的内容为 1AH(这些值是按正常的而不是按反向字节顺序表示的)。计算(a)栈顶(当前单元)的地址和(b)要执行的指令的地址。

2-12. 计算以下被访问数据的存储器地址。(a)DS 内容为 7E9B[0], 并且把数据从存储器传送到 AX 的指令是 MOV AX, [24H]。(b)DS 的内容为 43C6[0], 并且把数据从存储器传送到 BX 的指令是 MOV BX, [48H]。

计算机存储器与执行指令

第 3 章

目的：介绍程序向存储器的输入并跟踪它们的执行情况。

3.1 引言

本章使用叫做 DEBUG 的 DOS 程序(它允许你查看存储器)把程序输入到存储器中，并跟踪它们的执行情况。书中讲述了怎样把程序直接输入到代码段中，同时提供了对每个执行步骤的说明。虽然有不少完善的调试程序，例如 CODEVIEW 和 TurboDebugger，我们还是使用 DEBUG，原因在于它用起来既简单又通用。

最初的练习说明如何检查存储器的特定区域的内容。第一个程序例子使用“立即”数据，它是在指令中定义的，用于向寄存器装入数据并执行算术运算。第二个程序例子使用分别由可执行指令定义的数据。伴随指令执行去跟踪它们，可以对程序的操作和各种寄存器的作用有更加深入的了解。

你可以在没有汇编语言预备知识、甚至没有程序设计预备知识的情况下马上开始工作了。只需一台基于 Intel 的 PC 和 DOS 系统(独立的或是 Windows 下的)。不过，我们还是得假定你对系统命令、选择磁盘驱动器与文件比较熟悉。

3.2 使用 DEBUG 程序

DEBUG 程序是用来测试和调试可执行程序的。它能显示所有的程序代码和十六进制格式的数据，以及输入存储器的任何数据(也必须是十六进制格式)。DEBUG 还提供了一种单步方式，这种方式允许一次执行一条指令，这样就能查看每条指令在存储单元和寄存器中的结果。

DEBUG 命令

DEBUG 的一组命令能完成许多有用的操作。常用命令如下：

A 把符号指令汇编成机器码

- D 以十六进制格式显示一个存储区的内容
- E 从一个指定的存储单元开始, 把数据输入到存储器中
- G 运行在存储器中的可执行程序(G 的意思是“go”)
- H 完成十六进制算术运算
- N 给程序命名
- P 执行一组相关指令
- Q 退出 DEBUG
- R 以十六进制格式显示一个或多个寄存器的内容
- T 跟踪一条指令的执行
- U 把机器码反汇编为符号码

DEBUG 命令的规则

这里是一些使用 DEBUG 的基本规则:

- DEBUG 不区分大小写字母, 可以用任何一种方法输入命令。
- DEBUG 假定所有数都是十六进制格式的。
- 命令中的空格只是用来分隔参数的。
- 段和偏移值是用冒号来指定的, 形如“段基址: 偏移值”。

附录 C 提供全部 DEBUG 命令的完整说明, 包括有关如何启动它。现在就让我们利用 DEBUG 命令在存储器中做一番“游历”。

DEBUG 显示命令

D 命令把指定数据区内容显示在屏幕上。下面 3 个例子使用 DEBUG 的 D 命令显示存储器的同一个区域, 这个区域从数据段(DS)的偏移值 200H 处开始:

```
D DS:200          (大写字母的命令, 跟着空格)
DDS:200          (大写字母的命令, 未跟着空格)
dds:200          (小写字母的命令, 未跟着空格)
```

显示的屏幕由 3 部分组成:

1. 左边是最左边所显示的字节的十六进制地址, 其格式为: 段: 偏移值。
2. 中央宽的区域是所显示区域的十六进制表示。
3. 右边是字节的 ASCII 表示, 它包含可显示的字符, 这些字符解释十六进制区域。

D(显示)命令显示 8 行数据, 每行包含 16 个字节(32 个十六进制数字), 一共是 128 个字节, 从指定的地址开始。

地址	←----- 十六进制表示 -----→	←----- ASCII -----→
xxxx:xx10	xx.....xx-xx.....xx	x.....x
xxxx:xx20	xx.....xx-xx.....xx	x.....x
xxxx:xx30	xx.....xx-xx.....xx	x.....x
...		
xxxx:xx80	xx.....xx-xx.....xx	x.....x

左边的地址只是最左边(起点)字节的地址,其格式是段:偏移值,可以跨行计数去确定每个其他字节的位置。十六进制表示的区域把每个字节表示成2个十六进制字符,为清晰起见,用一个空格跟着。另外,还是为清晰起见,第二个8个字节与第一个8个字节之间用一个短线加以分隔。这样,如果你要去定位在偏移值 `xx13H` 处的字节,那么就从 `xx10H` 开始,并逐次向右数3个字节即可。

D 命令还能列出寄存器的内容和标志寄存器的状态。

3.3 查看存储单元

最初的两个练习涉及使用 DEBUG D(显示)命令去查看所选择的存储单元的内容。

3.3.1 练习 1: 检查 BIOS 数据区

第一个练习是检查在存储器低端、起点在 `400H` 单元(或更精确地说是段地址 `40[0]`)的 BIOS 数据区的内容。当计算机接通电源后, BIOS 初始化这个区域的值并在程序执行过程中修改它们。

可以用一个两部分组成的地址查看这些值:一部分是段地址 `40`(技术上说是 `400`, 因为最后的 `0` 被移出了);另一部分表示与段地址间距离的偏移值 `nn`。把地址 `40:nn` 理解为段 `40[0]H` 加上偏移值 `nnH`。在 24.3 节“BIOS 数据区”会对这些数据项做更详细的解释。

1. 检查串行与并行端口

BIOS 数据区的第一个 16 字节包含串行与并行端口的地址。准确地键入:

D 40:00 (并按<Enter>键)

所显示的第一个 4 个字表示串行端口 COM1 到 COM4。如果有两个串行端口,则第一个双字是以相反字节顺序排列的 `F803` 和 `F802`, 端口是在 `03F8` 和 `02F8`。第二个 4 个字表示并行端口 LPT1 到 LPT4。对于有一个并行端口的系统,第一个字很可能是 `7803`, 它对应于端口 `0378`。

2. 检查系统设备

BIOS 数据区里的设备状态字提供已安装设备的简单的指示。可以用 DEBUG 命令把这个字定位在 `410H-411H` 单元, 这条命令是:

D 40:10 (并按<Enter>键)

显示结果如下:

0040:0010 xx xx ...

假定在设备状态字中的两个字节是十六进制值 `23` 和 `44`。为解释它们,把字节反向排列为 `4423` 并把它们转换成二进制格式:

二进制值: 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1
位位置: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

下面是对这些位的说明:

位	设备
15, 14	所连接的并行打印机端口数=1(二进制 01)
11-9	所连接的串行端口数=2(二进制 010)
7, 6	软盘设备数=1(00=1, 01=2, 10=3, 11=4)
5, 4	初始显示方式=10(01=40×25 彩色, 10=80×25 彩色, 11=80×25 单色)
1	1=数值协处理器存在
0	1=软盘驱动器存在
不使用其他未涉及到的位。	

3. 检查键盘 Shift 状态

位于 BIOS 数据区 417H 单元的是键盘 Shift 状态的第一个字节。首先要确认 NumLock 和 CapsLock 是关闭的, 然后用命令 D 去查看 40:17 单元, 显示如下开始:

```
0040:0017 00 00 ...
```

现在打开两个 Lock 键并重复显示命令, 40:17 中的值应该是 60 00。

4. 检查显示状态

位于 BIOS 数据区 449H 单元的是第一个显示数据区。键入命令 D 40:49。第一个字节包含当前显示方式(如 03 表示彩色), 而第二个字节是屏幕上的列数(这里是 50H=80)。你还能找到位于 40:84H 单元的行数。

3.3.2 练习 2: 考察 ROM BIOS

以下练习检查在存储器高端的 ROM BIOS 中的数据。

1. 检查版权通告与系列号

计算机的版权通告是嵌入在 ROM BIOS 中的 FE00H 单元的。为了查看这一段, 可以键入 D FE00:0。由计算机制造时间决定, 紧跟版权通告之后, 在一般机器上还有 7 位数字的系列号。版权通告更多地可以从屏幕右边 ASCII 区域的字符中观察到, 而系列号则按十六进制数加以辨认。版权通告可能比较长, 超出了已经显示的部分, 为继续查看它, 可以简单地再按一次 D 键并跟着按一下回车(<Enter>)键。

2. 检查 ROM BIOS 数据

ROM BIOS 的制造日期是按 mm/dd/yy 记录的, 起点在 FFFF5H 单元。需要这一段时, 键入 D FFFF:5。了解这一日期对于确定计算机的年代和型号是有用的。

现在你已经知道如何使用显示命令了, 可以查看任何存储单元的内容, 还可以步进地从头到尾查看存储器, 这只要简单地重复按 D 键即可——DEBUG 依次显示 8 行(128 个字节), 直到最后的 D 操作为止。

当你完成上述试验后, 键入 Q(用于退出)从 DEBUG 退出, 或者继续进行下面的练习。

3.4 机器语言举例 1: 使用立即数据

现在让我们使用 DEBUG 把两个程序中的第一个程序直接输入到存储器中, 并跟踪它的执行情况。两个程序都说明简单的机器语言指令在存储器中的出现和它们执行的效果。为了这一目的, 我们将从使用 DEBUG E(Enter)命令开始。要特别注意, 在一个错误单元输入数据或者输入错误的数据可能会产生无法预料的后果, 好像并没有造成任何破坏, 但你可能得到意外的一个位, 也可能丢失在 DEBUG 对话期间所输入的数据。

第一个程序使用**立即数据**——数据被定义为指令的一部分。下面所表示的是十六进制格式的机器语言, 为清晰起见而与注释合在一起的符号码。对于第一条指令——B82301, 符号码是 MOV AX, 0123, 它把值 0123H 传送到 AX(你定义一个正常的而不是反向字节顺序的立即值)。MOV 是指令, AX 是第一个操作数, 而立即值 0123H 是第二个操作数。

机器指令	符号码	注释
B82301	MOV AX, 0123	把值 0123H 传送到 AX
052500	ADD AX, 0025	把值 0025 加到 AX 中
8BD8	MOV BX, AX	把 AX 的内容传送到 BX
03D8	ADD BX, AX	把 AX 的内容加到 BX
8BCB	MOV CX, BX	把 BX 的内容传送到 CX
2BC8	SUB CX, AX	从 CX 中减去 AX 的内容
2BC0	SUB AX, AX	从 AX 中减去 AX (清除 AX)
EBEE	JMP 100	返回到起点

注意: 机器指令的长度可能是一个、两个或三个字节。第一个字节是实际操作, 而存在的任何其他字节都是**操作数**——访问的立即值, 寄存器, 或存储单元。程序是从第一条机器指令开始执行的, 并且顺序地、一个接一个地步进执行每条指令。现在不要求弄清楚机器码的意思。例如, 在一种情况下, MOV 的机器码(第一个字节)是 hex B8, 而在另一种情况下, MOV 的代码是 hex 8B。

3.4.1 键入程序指令

就像你前面所做的那样, 开始进行这个练习: 键入命令 DEBUG 并按回车键。当 DEBUG 全部装入完毕, 会显示它的提示符(-)。为了把这个程序直接键入到存储器中, 只是输入机器指令部分, 而不是符号码或注释。键入以下的 E 命令(包括空格), 表示为:

```
E CS:100 B8 23 01 05 25 00 (并按<Enter>键)
```

CS:100 指明要存放数据的存储器起始地址是在代码段起点之后 100H(256)字节处(在 DEBUG 下, 机器码的正常起始地址)。E 命令使 DEBUG 把每一对十六进制数字存入存储器中的一个字节, 从 CS:100 到 CS:105。

下一条 E 命令存放 6 个字节, 从 CS:106 到 CS:10B:

E CS:106 8B D8 03 D8 8B CB (并按<Enter>键)

最后一条 E 命令存放 6 个字节, 从 CS:10C 到 111:

E CS:10C 2B C8 2B C0 E8 EE (并按<Enter>键)

如果你键入了错误命令, 那么就用正确的值简单地重复做一下。

3.4.2 执行程序指令

现在要做每次执行一条上述指令是一件简单事情。图 3-1 表明了整个步骤, 包括使用 E 命令去键入机器码。屏幕应该显示出和你输入的每条 DEBUG 命令类似的结果。在每条指令执行完之后, 还可以查看寄存器的内容。这里, 和我们有关的新的命令是 R(寄存器)和 T(跟踪)。

为了查看寄存器和标志的初始内容, 可以键入 R 命令, 接着按回车键, 如图 3-1 的第 4 行所示。DEBUG 以十六进制格式显示寄存器的内容, 如:

AX=0000 BX=0000 ...

由于计算机配置的不同, 在屏幕上的某些寄存器内容也将与图 3-1 所示的不同。DEBUG 已经把 DS、ES、SS 和 CS 全部用同样的段地址 xxxx[0]初始化了。IP 应该显示 IP=0100, 它指明指令的执行是从代码段起点后的 100H 个字节开始的(这就是你要用 E CS:100 作为程序起点的原因)。

图 3-1 中的标志寄存器指明以下对于溢出、方向、中断、符号、零、辅助进位、奇偶性以及进位标志的初始设置:

NV UP EI PL NZ NA PO NC

这些设置的意思分别是无溢出, 方向向上(或向右), 允许中断, 符号为正, 非零, 无辅助进位, 奇的奇偶性, 以及无进位。此时, 这些设置对我们来说都是不重要的。

紧随寄存器并同样用 R 命令显示的, 是第一条要执行的指令。注意, 在图中 CS 寄存器的内容是 21C1。由于 CS 段地址肯定和这个值不同, 所以在指令中把它表示为 xxxx:

xxxx:0100 B82301 MOV AX, 0123

- xxxx 指明代码段的起点是 xxxx[0]。值 xxxx:0100 指的是在 CS 段地址 xxxx[0]之后有 100 个字节的偏移。
- B82301 是 CS:100 输入的机器码。
- MOV AX, 0123 是符号汇编指令, 由 DEBUG 根据机器码确定。实际上, 这条指令的意思是: 把立即值 0123H 传送到 AX。DEBUG 把机器指令进行“反汇编”, 以便更容易解读它们。在本章之后, 你将专用符号汇编指令编码。

请记住这里的警告: 在试图执行一条指令之前, 一定要确保它是有效的。即使你键入了一个不正确的机器码, DEBUG 还是会试图去执行它, 而且假如这个代码是无效的, 处理器就会死锁, DEBUG 无法工作下去, 系统只能重新引导。

为了执行 MOV 指令, 键入 T(跟踪)命令并按<Enter>键。机器码是 B8(传送到 AX)后面跟着 2301。该操作把 23 传送到 AX 的低半部分(AL), 并把 01 传送到 AX 的高半部分(AH):

	AH	AL
AX:	01	23

```

-E CS:100 B8 23 01 05 25 00
-E CS:106 8B D8 03 D8 8B CB
-E CS:10C 2B C8 2B C0 EB EE
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0100 NV UP EI PL NZ NA PO NC
21C1:0100 B82301 MOV AX,0123
-T
AX=0123 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0103 NV UP EI PL NZ NA PO NC
21C1:0103 052500 ADD AX,0025
-T
AX=0148 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0106 NV UP EI PL NZ NA PE NC
21C1:0106 8BD8 MOV BX,AX
-T
AX=0148 BX=0148 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0108 NV UP EI PL NZ NA PE NC
21C1:0108 03D8 ADD BX,AX
-T
AX=0148 BX=0290 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=010A NV UP EI PL NZ AC PE NC
21C1:010A 8BCB MOV CX,BX
-T
AX=0148 BX=0290 CX=0290 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=010C NV UP EI PL NZ AC PE NC
21C1:010C 2BC8 SUB CX,AX
-T
AX=0148 BX=0290 CX=0148 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=010E NV UP EI PL NZ AC PE NC
21C1:010E 2BC0 SUB AX,AX
-T
AX=0000 BX=0290 CX=0148 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0110 NV UP EI PL ZR NA PE NC
21C1:0110 EB EE JMP 0100

```

图 3-1 跟踪机器指令执行

DEBUG 显示寄存器中的操作结果。IP 寄存器现在的内容是 0103H(原来的 0100H 加上第一个机器码指令长度的 3 个字节), 0103H 这个值指明代码段中下一条要执行指令的偏移地址, 即:

```
xxxx:0103 052500 ADD AX, 0025
```

为了执行这条 ADD 指令, 要输入另一个 T 命令。该指令把 25H 加到 AX 的低半部分(AL), 并把 00H 加到 AX 的高半部分(AH), 实际上是把 0025H 加到 AX 中。现在 AX 的内容是 0148H, 而 IP 的内容是 0106H, 用于下一条要执行的指令:

```
xxxx:0106 8BD8 MOV BX, AX
```

键入另一个 T 命令。MOV 指令把 AX 的内容传送到 BX。注意: 传送之后, BX 的内容是 0148H, AX 仍然是 0148H, 因为 MOV 是复制, 而不是真的把数据从一个地方移到另一个地方。

现在连续键入 T 命令, 一步步地执行余下的各条指令。ADD 指令把 AX 的内容加到 BX, 在 BX 中得到 0290H。然后, 程序把 BX 的内容传送(复制)到 CX, 从 CX 中减去 AX, 并且从 AX 本身再减去 AX。最后的操作是把 AX 清除为零并把零标志从 NZ 改为 ZR(零), 以便指明操作的结果。

JMP 指令把 IP 复位成 100H, 使得正在进行的过程“跳转”返回到程序的起点。这里,

它是作为一种预防措施,因为在最后输入的指令之后是“无用信息”,这时如果试图去执行它,将会造成处理器死锁。注意,已被写入的程序是一个无限的循环,也就是说,没有办法结束它,尽管 DEBUG 还让它循环地工作。这类处理过程通常用在特殊的环境中,比如监控系统。

另外,图 3-1 还表明 DS、ES、SS 和 CS 全部含有同样的段地址。这是因为 DEBUG 会把全部段当作.COM 程序,用数据、堆栈和代码全在同一段内的办法来处理,尽管你为它们留有各自独立的段。当编写.EXE 程序时,要把这 3 个区域保存在各自独立的段内,每个段都在它自己的段地址之下。

为了返回这一程序,可以简单地按 T 键去执行 JMP 指令,它把 IP 复位成 100H,即回到程序的起点。

3.4.3 显示存储器内容

为了查看在代码段中的机器语言程序,要输入如下命令:

D CS:100

图 3-2 表示这条命令的结果,每行显示 16 个字节(32 个十六进制数字)的数据。右边是每个字节的 ASCII 表示(如果是标准字符)。在机器码的情况下,ASCII 表示法是没有意义的,可以忽略不管。下面几节将更详细地讨论显示的右边部分。

显示的第一行是从代码段的偏移值 100H 开始的,并且表示出了单元 CS:100 到 CS:10F 的内容。第二行表示的是 CS:110 到 CS:11F 的内容。实际上,虽然程序是在 CS:111 结束,但 D 命令会自动地显示从 CS:100 到 CS:170 的 8 行。在这个例子中,紧跟 CS:111 的任何数据都是“无用信息”。只是希望从 CS:100 到 CS:111 的机器代码和你自己显示的是一样的,随后的那些字节的内容是无用的。

输入 Q 命令,结束 DEBUG 对话,或继续进行下一个练习。

```
-D CS:100
21C1:0100  B8 23 01 05 25 00 8B D8-03 D8 8B CB 2B C8 2B C0  .#.%.....+..
21C1:0110  EB EE 8D 46 14 50 51 52-FF 76 28 E8 74 00 8B E5  ...F.PQR.v(.t...
21C1:0120  B8 01 00 50 FF 76 32 FF-76 30 FF 76 2E FF 76 28  ...P.v2.v0.v.v(
21C1:0130  E8 88 15 8B E5 FF 36 18-12 FF 36 16 12 8B 76 28  ....6...6...v(
21C1:0140  FF 74 3A 89 46 06 E8 22-CE 8B E5 30 E4 3D 0A 00  .t:.F.."...0.=..
21C1:0150  75 32 A1 16 12 2D 01 00-8B 1E 18 12 83 DB 00 53  u2.....S
21C1:0160  50 8B 76 28 FF 74 3A A3-16 12 89 1E 18 12 E8 FA  P.v(.t:.....
21C1:0170  CD 83 E5 30 E4 3D 0D 00-74 0A 83 06 16 12 01 83  ...0.=.t.....
```

图 3-2 代码段的转储

3.5 机器语言举例 2: 使用定义的数据

上一个例子是在 MOV 和 ADD 指令中使用直接定义的立即值。下一个例子是在程序中定义值(或常数)为 0123H 和 0025H 的数据项,程序指令去访问含有这些值的存储单元。

通过这个例子,可以深入了解计算机是如何借助于在 DS 寄存器中的地址和偏移地址来

存取数据的。这个例子定义以下的数据项，它起始于偏移值 0200H，这样可以清楚地和在 0100H 的指令分隔开来：

DS 偏移值	十六进制内容
0200H	2301H
0202H	2500H
0204H	0000H
0206H	2A2A2AH

请记住，一个十六进制数字占有半个字节，使得比如 23H 就被存放在数据区的偏移值 0200H(第一个字节)内，而 01H 则被存放在偏移值 0201H(第二个字节)内。下面是处理这些数据项的机器指令，数据项是按相反字节顺序输入的值(比如 0200 成为 0002)：

指令	注释
A10002	把起始于 DS 偏移值 0200H 的一个字(2 个字节)传送到 AX 中。
03060202	把起始于 DS 偏移值 0202H 的字(2 个字节)内容加到 AX。
A30402	把 AX 的内容传送到起始于 DS 偏移值 0204H 的字中。
EBF4	跳转到程序的起点。

请注意二条传送指令有不同的机器码：A1 和 A3。实际的机器码取决于所访问的寄存器、数据的大小(字节或字)、数据的传送方向(从寄存器来或到寄存器去)以及访问的是立即数据、存储器还是寄存器。

3.5.1 键入程序指令与数据

再一次使用 DEBUG 键入程序并跟踪它的执行。首先，使用 E 命令键入指令，起点在 CS：0100：

```
E CS:100 A1 00 02 03 06 02 02(按 <Enter> 键)
E CS:107 A3 04 02 EB F4(按 <Enter> 键)
```

使用 E 命令定义数据，假设从 DS：0200 开始：

```
E DS:0200 23 01 25 00 00 00(按 <Enter> 键)
E CS:0206 2A 2A 2A(按 <Enter> 键)
```

第一条 E 命令在偏移值为 0200 的数据区起点上存放 3 个字(6 个字节)，必须按相反字节顺序键入每一个字，使 0123 变成 2301，而 0025 变成 2500。当 MOV 指令顺序地取出这些字并把它们装入到寄存器中时，再把字节顺序反过来，使 2301 变成 0123，而 2500 变成 0025。

第二条 E 命令存放 3 个星号(***)，它们被定义为 2A2A2A，以便以后可以使用 D(显示)命令查看它们，其实这些星号在数据区里是没有什么特别用处的。

图 3-3 表示在程序中包括 E 命令在内的所有步骤，屏幕会显示类似的结果，注意在 CS 与 DS 中的地址可能不同。为了考察存放的数据(从 DS：200H 到 208H)和指令(从 CS：100H 到 10AH)，键入以下 D 命令：

```
查看代码: D CS:100, 10B (按 <Enter> 键)
查看数据: D DS:200, 208 (按 <Enter> 键)
```

检查 2 个所显示的区域的内容和图 3-3 所示的什么内容是相同的。

```

-E CS:100 A1 00 02 03 06 02 02
-E CS:107 A3 04 02 EB F4
-E DS:200 23 01 25 00 00 00
-E DS:206 2A 2A 2A
-D CS:100,10A
21C1:0100 A1 00 02 03 06 02 02 A3-04 02 EB F4 .....
-D DS:200,208
21C1:0200 23 01 25 00 00 00 2A 2A-2A # %...***
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0100 NV UP EI PL NZ NA PO NC
21C1:0100 A10002 MOV AX,[0200] DS:0200=0123
-T

AX=0123 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0103 NV UP EI PL NZ NA PO NC
21C1:0103 03060202 ADD AX,[0202] DS:0202=0025
-T

AX=0148 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=0107 NV UP EI PL NZ NA PE NC
21C1:0107 A30402 MOV [0204],AX DS:0204=0000
-T

AX=0148 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=21C1 ES=21C1 SS=21C1 CS=21C1 IP=010A NV UP EI PL NZ NA PE NC
21C1:010A EB F4 JMP 0100
-D DS:0200,0208
21C1:0000 23 01 25 00 48 01 2A 2A-2A #,%H.***
-Q

```

图 3-3 跟踪机器指令

3.5.2 执行程序指令

键入指令之后,就可以像以前所做的那样来执行它们了。首先,要确定 IP 的内容为 100H。然后,按 R 键去观察寄存器与标志的内容并显示第一条指令。虽然 AX 可能仍然包含来自以前练习的值,但你可以立刻取代它。第一条显示的指令是:

```
xxxx:0100 A10C02 MOV AX, [C200]
```

CS:0100 访问第一条指令 A10002。DEBUG 把这条指令解释成 MOV 并决定对数据区的第一单元[0200H]进行访问。方括号告诉你这次访问的是存储器地址而不是一个立即值(把立即值 0200H 传送到 AX 应当表示为 MOV AX, 0200)。

现在键入 T(跟踪)命令。指令 MOV AX, [0200]把在偏移为 0200H 中的字内容传送到 AX。该内容是 2301H,在按反向顺序操作送到 AX 中成为 0123H,并由它取代了 AX 以前的任何内容。

键入另一条 T 命令去执行下一条 ADD 命令。该操作把 DS:0202 中的字的内容加到 AX。AX 中的结果现在是 0123H 与 0025H 的和,即 0148H。

下一条指令是 MOV [0204], AX。为执行这条指令,键入 T 命令。该指令是把 AX 内容(0148H)复制到 DS 偏移值 204H 与 205H 处的数据区,反向排列为 4801H。为查看从 200H 到 208H 数据内容的变化,键入:

```
D DS:200,208 (按〈Enter〉键)
```

所显示的值为：

数据区：	23	01	25	00	48	01	2A	2A	2A
偏移值：	200	201	202	203	204	205	206	207	208

显示的左边表示的是出现在存储器中的实际机器码，右边简单地帮助你更容易定位字符数据。注意：这些表示在屏幕右边的十六进制值是它们的 ASCII 等效值。23H 与 25H 分别显示为一个符号(#)和一个百分号(%), 而 3 个 2AH 字节则产生星号(*)。

现在可以输入 Q，结束 DEBUG 对话，或者继续下一个练习。

3.5.3 重新执行指令

有时会需要复位 IP 寄存器的值，具体操作如下：

1. 键入 R IP，显示 IP 的内容。
2. 输入值 100 (或另一条指令的地址)，按回车键。

这一过程使你返回到程序(或程序中一条指令)的起点，现在就可以重复以前的步骤。键入 R 命令(不跟 IP)，DEBUG 显示寄存器、标志以及要执行的第一条指令。现在，你可以使用 T 命令再去跟踪指令的步骤。如果你的程序累计总数，则用 E 命令去清除各存储单元并用 R 命令去清除各寄存器。但是要确保 CS、DS、SP 和 SS 的内容不变，这些寄存器都是有专门用途的。

3.5.4 用 DEBUG 保存程序

在两种情况下，可以使用 DEBUG 把程序保存在磁盘上：

1. 从磁盘上取回一个已有的程序，加以修改，然后再保存它。
2. 使用 DEBUG 创建一个非常小的机器语言程序，并保存这个程序。

要做详细了解，可以参阅附录 C 中的 W(写)命令。

现在你可能发现 DEBUG 的 H(十六进制)命令对加减十六进制值是有用的。最大字段长度是 4 个十六进制数字。键入该命令，例如 H 3443 2A2B。该操作把和显示在左边，并把差显示在右边，如 5E6E 0A18。

3.6 一个汇编语言程序

虽然这里的程序举例都是机器语言格式的，我们也可以使用 DEBUG 去键入汇编语言语句。现在让我们试验一下用于把汇编语句输入到计算机中去的 DEBUG A 和 DEBUG U 命令。

3.6.1 A(汇编)命令

A 命令告诉 DEBUG 开始接收符号汇编指令并把它们转换成机器语言。初始化指令的起

始地址，它位于代码段偏移值 100H 处。如下所示：

```
A 100 <Enter>
```

DEBUG 显示代码段的地址和偏移值(0100)为 xxxx:0100。键入以下命令，每条指令后跟一个 <Enter>：

```
MOV CL, 42      <Enter>
MOV DL, 2A      <Enter>
ADD CL, DL      <Enter>
JMP 100         <Enter>
```

当键入完程序，再一次按 <Enter> 键，从 A 命令中退出。这是一个额外的 <Enter>，它告诉 DEBUG 已经没有更多的符号指令输入了。完成后，DEBUG 会显示每条指令的偏移地址：

```
xxxx:0100  MOV CL, 42
xxxx:0102  MOV DL, 2A
xxxx:0104  ADD CL, DL
xxxx:0106  JMP 100
xxxx:0108
```

在执行此程序之前，我们使用 DEBUG 的 U(反汇编)命令来检查一下所生成的机器语言。

3.6.2 U(反汇编)命令

DEBUG 的 U 命令把汇编语言指令显示成机器码，可以使用这条命令查看第一条与最后一条指令的存储单元的位置。在这里，它们是 100H 和 107H。键入：

```
U 100, 107 <Enter>
```

屏幕显示的各列是存储单元、机器码以及符号码，如下所示：

```
xxxx:0100  B142  MOV CL, 42
xxxx:0102  B22A  MOV DL, 2A
xxxx:0104  00D1  ADD CL, DL
xxxx:0106  E8F8  JMP 100
xxxx:0108
```

现在跟踪程序机器码的执行。从键入 R 去显示寄存器(IP 的内容应该是 0100)和第一条指令(MOV CL, 42)开始，然后用相继的 T 命令去跟踪随后的指令。当到达 JMP 时，IP 的内容应该是 106H，而且 CL 应该是 6CH。继续进行下一个练习或按 Q 键退出执行。

现在你已经了解到如何键入机器语言和汇编语言程序了。顾名思义，DEBUG 是为调试程序而设计的，而大多数工作涉及的将是常规汇编语言的使用，与 DEBUG 无关。

3.7 使用 INT 指令

以下 4 个例子说明如何请求有关系统的信息。使用 INT(中断)指令退出程序，进入 DOS 或 BIOS 例行程序，实现所需功能后，再返回程序。有不同类型的 INT 操作，其中的某些操

作需要在 AH 寄存器中的功能码去请求指定的动作。我们将使用 P(继续进行)命令去执行全部中断例行程序,而不是用单步操作的 T 命令。必须确保 IP 复位成 100H。

3.7.1 获得当前日期和时间

访问当前日期的指令是 INT 21H, 功能码 2AH。键入 DEBUG 命令 A 100, 然后键入以下汇编指令:

```
MOV AH, 2A    <Enter>
INT  21      <Enter>
JMP  100     <Enter>, <Enter>
```

键入 R 显示寄存器,并键入 T 执行 MOV,再键入 P 直接运行整个中断例行程序,该操作停在 JMP,寄存器包含以十六进制格式表示的这一信息:

```
AL:  星期, 其中 0=星期日
CX:  年(例如, 07D4H=2004)
DH:  月(01H 到 0CH)
DL:  日(01H 到 1FH)
```

访问当前时间的操作是 INT 21H 功能码 2CH。首先用 R IP 把 IP 复位为 100, 然后键入 DEBUG A 命令和以下汇编指令:

```
MOV AH, 20    <Enter>
INT  21      <Enter>
JMP  100     <Enter>, <Enter>
```

接着发生的过程和为取得日期所做过的事情是一样的。该操作把小时送给 CH(在 24 小时格式下, 00=午夜), 把分钟送给 CL, 把秒送给 DH, 而把百分之一秒送给 DL。

按 Q 键退出, 或者继续进行下一个练习(把 IP 复位为 100)。

3.7.2 确定安装的设备

前面的练习已经检查了与计算机所包含的设备有关的 401H 和 411H 单元。BIOS 还提供一中断例行程序 INT 11H, 它传送信息给 AX。键入 DEBUG 命令 A 100, 然后键入下面的指令:

```
INT 11      <Enter>
JMP 100     <Enter>, <Enter>
```

键入 R, 显示寄存器和第一条指令。INT 11H 这条指令传送控制给 BIOS 中的例行程序, 该例行程序把设备数据传送给 AX。重复按 T 键和回车键, 观察每条 BIOS 指令的执行情况 (这一过程违反了不跟踪整个中断过程的规则, 但这一操作还是正确的)。

在 BIOS 中的实际指令可能和这里的指令稍微有些不同, 这取决于所安装的版本(右边的注释是作者加的):

```
JMP EE53      :
PUSH DS       : 把 DS 地址保存在堆栈中
```

```

MOV AX, 0040      ; 得到段地址
MOV DS, AX        ; 把它送到 DS
MOV AX, [0010]    ; 把从 40:10 中得到的数据送到 AX
POP DS            ; 恢复 DS 中的地址
IRET              ; 从中断返回

```

最后的 T 命令从 BIOS 退出并返回到 DEBUG。如果刚才 BIOS 中的执行没有出错，现在 AX 中就包含了已安装设备的记录。现在显示的指令是你所输入的 JMP。按 Q 键退出，或者继续进行下一个练习(把 IP 复位为 100)。

3.7.3 使用 INT 完成显示

这个在屏幕上显示数据的练习引入一些新的特性，键入 DEBUG 命令 A 100 和以下一些汇编指令：

```

100  MOV AH, 09
102  MOV DX, 109
105  INT 21
107  JMP 100
109  DB 'your name', '$' <Enter>, <Enter>

```

两条 MOV 指令告诉 INT 21H 要进行显示(AH=09)和从什么起始地址(DX=109)开始。注意，偏移值 109 开始定义你的名字，其中 DB 指的是“定义字节”，而字符则包含在单引号中。你的名字(your name)之后是美元符号(\$)，它也在引号中，是用来通知 INT 结束显示的。

键入 R 显示寄存器和第一条指令，然后为 2 条 MOV 指令键入 T 命令。键入 P 去执行 INT 21H，将会看到所显示的“your name”。按 Q 退出，或者继续进行下一个练习(把 IP 复位为 100)。

3.7.4 使用 INT 进行键盘输入

这个从键盘上接受字符的练习又引入了一些新的特性。键入 DEBUG 命令 A 100 和以下这些汇编指令：

```

100  MOV AH, 10
102  INT 16
104  JMP 100      (Enter), (Enter)

```

第一条 MOV 指令提供了功能码 10H，它通知 INT 16H 从键盘上接收数据，该操作把来自键盘的字符传送到 AL 寄存器。键入 R 显示寄存器和第一条指令，然后键入 T 命令执行该指令。当为 INT 16H 键入 P 时，系统会等待你按键。如果你按的是 1 键，那么会看到该操作把 31H(ASCII 1 的十六进制表示)送到 AL。键入 T 去执行 JMP 指令，将返回到在 100 处的 MOV。使用 T 去执行 MOV。当为 INT 键入 P 时，系统会再一次等待你按键。如果你按 2，就可以看到该操作把 32H 送到 AL。你可以不受限制地继续像这样做下去。可以得到在附录 F 中的键盘 ASCII 码的表。按 Q 键退出或继续进行下一个练习(把 IP 复位成 100)。

3.8 使用 PTR 操作符

下一个程序举例引入了某些新的特性。在这个例子中，你会在寄存器与存储单元之间进行数据的传送与相加。在以前的程序中，把数据传送到寄存器，DEBUG 能从寄存器(AL 或 AX)的长度来判断要传送多少字节。但是，这里的程序是要传送立即数据到存储器的。由于象 MOV [120], 25 这样的指令不能指明字节的数量，这时，你可以使用 PTR 操作符。下面是指令：

```

100  MOV  AX, [11A]
103  ADD  AX, [11C]
107  ADD  AX, 25
10A  MOV  [11E], AX
10D  MOV  WORD PTR [120], 25
113  MOV  BYTE PTR [122], 30
118  JMP  100
11A  DB   14 23
11C  DB   05 00
11E  DB   00 00
120  DB   00 00 00

```

以下是这些指令的说明：

100：把存储单元 11AH-11BH 的内容传送到 AX。方括号指出这是存储器地址，而不是立即值。

103：把存储单元 11CH-11DH 的内容加到 AX。

107：把立即值 25H 加到 AX。

10A：把 AX 的内容传送到存储单元 11EH-11FH。

10D：把立即值 25H 传送到存储单元 120H-121H。注意 WORD PTR 操作符的使用，这个操作符通知 DEBUG 值 25H 是传送给在存储器中的一个字。如果你编写的指令是 MOV [120], 25，则 DEBUG 将无法确定想要的长度并会显示 ERROR(错误)信息。尽管你很少需要使用 PTR 操作符，但是真需要它的时候，你就会理解它还是非常必要的。

113：把立即值 30H 传送到存储单元 122H。这时，使用 BYTE PTR 操作符去指明一个字节的长度。

11A：定义字节值 14H 和 23H。DB 告诉 DEBUG 为数据项“定义字节”，这些数据项是指令(比如在 100 的一条)所要访问的。

11C, 11E 和 120：定义程序中所用的其他字节值。

为了键入这一程序，首先键入 A 100 <Enter>，然后键入每条符号指令(但不要存储单元)。最后，键入一个附加的回车键，从 A 命令退出。

为了执行程序，开始先键入 R 显示寄存器和第一条指令，然后连续键入 T 命令。当到达

118 的 JMP 时, 停止执行。键入 D 110 去查看改变了的 AX 的内容(233E)以及存储单元 11EH-11FH(3E23)、120H-121H(2500)和 122H(30)的内容。键入 T 去重复该程序或键入 Q 退出该程序。

本章中已经涉及了许多资料, 这些资料通过复习可以理解得更加清楚。

3.9 要 点

- DEBUG 程序对于测试与调试机器语言程序和汇编语言程序是有用的。它的命令包括像显示、输入与跟踪这样一些有用的操作。
- 由于 DEBUG 对大、小写字母是不加区别的, 所以可以用二者之中任何一种方法输入命令。
- 在 DEBUG 中, 所有输入与显示的数都是十六进制的。
- 如果在数据段或代码段中输入一个不正确的值, 那就要重新输入 E 命令去更正它。
- 使用 T 命令执行单条指令, 使用 P 命令正确执行一个完整的 INT 操作。
- 为重新执行第一条指令, 要把指针寄存器 IP 设置成 0100。键入 R(寄存器)命令后跟一个选定的寄存器, 如 R IP, 然后按回车键, DEBUG 显示 IP 的内容并等待输入。键入值 100, 接着按回车键。

3.10 习 题

3-1. 说明以下 DEBUG 命令的用途:

(a) A, (b) U, (c) P, (d) T, (e) Q, (f) D, (g) R, (h) E。

3-2. 分别写出满足下列要求的 DEBUG 命令:

- (a) 显示所有寄存器的内容。
- (b) 显示 IP 寄存器的内容并把它的内容改变成 100H。
- (c) 显示在数据段中起始于偏移值 2BCH 的数据。
- (d) 显示起始于存储单元 3AFH 的数据(要求把这一地址分离成它的段和偏移值)。
- (e) 把在单元 100H 到 12BH 的符号码进行反汇编。
- (f) 把 24A63BH 键入数据段内起始于 18AH 的存储单元。

3-3. 为以下操作提供机器码指令(a)把十六进制立即值 03A8 加到 AX, (b)把十六进制值 2CA4 传送到 AX 寄存器。

3-4. 假设已用 DEBUG 输入了以下 E 命令:

```
E CS:100 B8 45 01 05 25 00
```

十六进制 45 应该是 54。编写另一个 E 命令去校正一个不正确的字节, 即直接把 45 变成 54。

3-5. 在文本方式下, 彩色显示器的视频显示区是在地址为 B800[0]处。(a)使用 DEBUG D 命令显示这个区域。注意: 视频区中的每个字符后面都跟着它的属性(07 表示黑白的)。使用

附加的单个 D 命令去显示更多的部分。(b)使用 DEBUG F(填充)命令用 50 个星号(2AH)部分地填充屏幕。例如,以下的命令用 4000(FA0H)个心形符号(03)与属性(这里,16H 是指在蓝色背景下的褐色字符的属性)填入整个视频区域:FB800:0 LFA0 03 16。

3-6. 假设已输入以下 DEBUG E 命令:

```
E CS:100 B8 05 1B 05 2C 00 EB F8
```

(a) 这里 3 条符号指令所表示的是什么?(参照本章第一个程序)

(b) 在执行这一程序时,你发现 AX 寄存器是以 4705 结束而不是预期的 0547,错误是什么?你准备如何去更正它?

(c) 更正指令后,现在想从第一条指令起重新执行该程序,所需的 DEBUG 命令是什么?

3-7. 考虑机器语言指令:

```
B0 1C D0 E0 B3 12 F6 E3 EB F6
```

这些指令完成以下操作:(a)把十六进制值 1C 传送到 AL 寄存器,(b)将 AL 的内容向左移一位(等于乘以 2),(c)把十六进制值 12 传送到 BL,(d)AL 乘以 BL。

使用 DEBUG 的 E 命令从 CS:100(记住这些是十六进制值)开始输入程序。下一步键入 D CS:100 去查看它。然后键入 R 和足够多相继的 T 命令,一步步地执行程序直到到达 JMP。AX 中的最终乘积是什么?

3-8. 使用 DEBUG 的 E 命令,输入以下机器语言程序:

```
机器码(在100H): A0 00 02 D0 E0 F6 26 01 02 A3 02 02 90
```

```
数据(在200H): 1E 16 00 00
```

该程序完成以下操作:(a)把 DS:200 处的字节内容(1E)传送到 AL 寄存器,(b)把 AL 的内容左移一位,(c)AL 乘以 DS:0201 处的一个字节内容(16),(d)把乘积从 AX 传送到起始于 DS:0202 的字中。

键入 D 命令,查看代码和数据。然后键入 R 和足够多相继的 T 命令,一步步地执行程序直到到达 JMP 为止。AX 中的最终乘积是什么?键入另一个 D DS:0200 并注意乘积是如何存放在 DS:0202 中的。

3-9. 根据习题 3-7 编写命令。该命令在用 HEXMULT.COM 的名字,把程序写入磁盘(见附录 C)。

3-10. 使用 DEBUG 的 A 命令,输入以下指令:

```
MOV DX, 2E
ADD DX, 1F
SHL DX, 01
SUB DX, BA
JMP 100
```

把这些指令反汇编并跟踪它们的执行情况,直到 JMP 为止,同时检查每条指令执行后的 DX 值。

3-11. 中断指令的用途是什么?

3-12. 使用 DEBUG 建立并运行一个程序。该程序显示短语“Coffee Break”。由 A 100 开始输入指令并为该短语使用 A 120(记住 \$ 定界符)。提示:参阅“使用 INT 完成显示”一节。

3-13. 使用 DEBUG 建立并运行一个程序。该程序从键盘接收 3 个字符并加以显示。(a)

由 A 100 开始。(b)使用 INT 16 接受一个字符到 AL 中并把该字符传送到单元[200]。(c)使用第二个 INT 16 接受一个字符到 AL 中并把该字符传送到单元[201]。(d)使用第三个 INT 16 接受一个字符到 AL 中并把该字符传送到单元[202]。(e)使用一个 E 123 ‘\$’ 命令在 3 个被存储的字符末尾定义一个 ‘\$’。(f)最后, 使用 INT 21 去显示这些字符。提示: 参阅 3.7.4 节 “使用 INT 进行键盘输入”。

!
:
:

第二部分

汇编语言的基础 知识

目的：阐述汇编语言程序与定义数据项的基本要求。

4.1 引 言

在第 3 章里，我们学习了如何利用 DEBUG 键入和执行机器语言程序。毫无疑问，机器码在理解方面是非常困难的，即使对于小程序也是一样。利用 DEBUG 的 A 命令键入一个小的汇编源程序，毫无疑问它比机器码更加容易理解。DEBUG A 命令的使用只是提供一种方便，从这一章起，你将开发更大的程序，这将需要更多的对于程序进行文档处理与修改的能力。

编写汇编程序要遵循一套严格的规则，首先使用编辑程序或字处理程序把它作为文件键入到计算机中，然后使用汇编编译程序去读取该文件并把它转换成机器码。

在这一章里，我们说明开发汇编程序的基本要求：注释的使用，通用的编码格式，控制汇编程序列表的伪操作，以及定义段与过程的要求。还会涉及程序的一般组织，包括初始化程序和结束程序的执行。最后要提及的是有关定义数据项方面的要求。

程序设计语言的两个主要级别是高级的和低级的。程序员用高级语言(如 C 或 BASIC)编写程序，使用强有力的命令，这些命令中的每一个都可以生成许多机器语言指令。另一方面，程序员用低级的汇编语言编写程序时，每条代码符号指令只能生成一条机器指令。虽然用高级语言编码更为富有成效，但是用汇编语言编码也有它的一些优点，大体有以下几方面：

- 在管理专用的硬件设备方面提供了更多的控制方法。
- 生成较小的、更加紧凑的可执行模块。
- 更快的执行效果。

通常的做法是把 2 种级别程序设计的好处结合起来：工程中的大部分使用高级语言编码，而一些关键性的模块(会产生显著延迟的)则用汇编语言编码。

不管你使用的是什么程序设计语言，它仍是符号语言，还必须把它翻译成计算机能够执行的形式。高级语言使用编译程序把源码翻译成机器码(称为目标码)。低级语言使用汇编程序把程序翻译成目标码。连接程序对高级语言与低级语言完成这样的过程：把目标码转换成可执行的机器语言。

4.2 汇编语言特性

首先涉及的汇编语言的一些特性是：程序注释，保留字，标识符，语句，以及伪操作。这些特性为语言提供了基本规则和框架。

4.2.1 程序注释

遍及整个程序的注释的使用可以改善程序的清晰度，特别是在汇编语言程序中更是如此，因为一组指令的目的往往是不清楚的。例如，指令 `MOV AH, 10H` 把 10H 传送给 AH 是显而易见的，但是这么做的原因可能并不清楚。注释用一个分号(;)开始，可以把它放在任何地方，汇编程序假定一行中所有在它右边的字符就是注释。注释可以包括任何可打印字符，连空格在内。

注释本身可以自成一行，如：

```
      ; Calculate productivity ratio(计算生产率)
```

或跟在指令之后的同一行里，如

```
ADD AX, BX      ; Accumulate total quantity (累计总量)
```

因为注释只出现在汇编的源程序列表中，而且不产生机器码，所以你可以有任意数量的注释而不会对汇编程序的大小和执行有影响。在本书中，所有汇编指令都是用大写字母，而所有注释都是用小写字母，这样做只是一种习惯并能使程序更具可读性。从技术上说，指令和注释可以随便使用大写或小写字母。提供注释的另一个途径是用 `COMMENT` 伪操作，这将在第 25 章加以讨论。

4.2.2 保留字

汇编语言里的某些名字是为它们固有的用途而保留的，只在特殊情况下使用。根据类型，保留字包括：

- 指令，如 `MOV` 和 `ADD`，这些是计算机可执行的操作。
- 伪操作，如 `END` 或 `SEGMENT`，可以利用它们为汇编程序提供信息。
- 操作符，如 `FAR` 和 `SIZE`，可以在表达式中使用。
- 预定义符号，如 `@ Data` 和 `@ Model`，它们在汇编期间向程序返回信息。

错误地使用了保留字，会使汇编程序产生出错信息。见附录 D 有关保留字的表。

4.2.3 标识符

标识符(或符号)是在程序中加入到所希望访问的项上的名字。有两类标识符：名字和标号；

1. 名字指的是一个数据项的地址，比如下列语句中的 `COUNTER`：

```
COUNTER DB 0
```

2. 标号指的是指令、过程或段的地址，比如下列语句中的 MAIN 和 B30:

```
MAIN PROC FAR
B30: ADD BL, 25
```

名字与标号的规则是一样的。标识符可以使用以下字符:

类型	容许的字符
字母表中的字母:	A 到 Z 与 a 到 z
数字:	0 到 9(不能是第一个字符)
专用字符:	问号(?)
	破折号, 或下划线(_)
	美元(\$)
	at(@)
	点或句号(.) (不能是第一个字符)

标识符的第一个字符必须是字母或专用字符, 但专用字符的点(.)除外。由于汇编程序使用一些 @ 打头的专用字, 所以应当避免把它用在你自己的定义中。

默认地, 汇编程序把大写和小写字母看成是相同的(命令行有一个选项, 该选项强制汇编程序对大小写敏感)。MASM 6.0 以前, 标识符的最大长度是 31 个字符, 此后它可以是 247 个。有效名字的例子是 TOTAL, QTY250, 以及 \$P50。推荐使用描述性的、有意义的名字。

寄存器的名字, 比如 AH、BX 和 DX, 它们是为访问这些寄存器而保留的。因此, 在一条指令中, 如 ADD CX, BX, 汇编程序知道 CX 和 BX 指的是寄存器。然而, 在一条 MOV REGSAVE, CX 的指令中, 只有你把 REGSAVE 定义为一个数据项的名字, 汇编程序才能识别它。

4.2.4 语句

汇编程序是由一组语句组成的。语句的两种类型是:

- (1) 指令类, 如 MOV 和 ADD, 汇编程序把它们翻译成目标码;
- (2) 伪操作类, 通知汇编程序完成特定的动作, 如定义数据项。

以下是语句的格式, 其中方括号指明可选的输入:

[identifier]	operation	[operand(s)]	[; comment]
--------------	-----------	--------------	-------------

标识符(identifier)(如果有的话), 操作(operation), 以及操作数(operand)(如果有的话)是用至少一个空格或制表符(tab)分隔开的。MASM 6.0 以前, 一行的最大字符数为 132 个, 此后可以是 512 个。然而大多数程序员宁愿把它限制在 80 个字符以内, 因为这是大多数屏幕可以容纳的最大数量。两个语句的例子如下:

	标识符	操作	操作数	注释
伪操作:	COUNT	DB	1	: 名字, 操作, 操作数
指令:	L30	MOV	AX, 0	: 标号, 操作, 2 个操作数

标识符, 操作, 以及操作数可以从任一列开始。但是, 为了这些输入项能产生更为可读的程序, 一般都是从同一列开始。此外, 许多编辑程序提供制表符(tab 键)让每 8 个位置断开一下,

以便于字段间留出空格。

正如在前面“标识符”标题下所说明的，术语**名字**是用作所定义的项目或伪操作的名字，而术语**标号**则是用作指令的名字。从现在起，我们将使用这些术语。

操作是语句中必须有的，最通常的用途是定义数据区和指令编码。对于数据项，如 **DB** 或 **DW** 那样的操作是定义字段、工作区或常数的。对于指令，如 **MOV** 和 **ADD** 那样的操作是指明要完成的动作。

操作数(如果有的话)提供操作要用的信息。对于数据项，操作数定义它的初始值。例如，在以下名字为 **COUNTER** 的数据项定义中，操作 **DB** 的意思是“定义字节”，操作数把它的内容初始化为零：

名字	操作	操作数	注释
COUNTER	DB	0	；用初始的零值定义字节

对于一条指令，操作数指明要完成的动作。指令的操作数可以是一个、两个或甚至于没有。以下是 3 个例子：

操作	操作数	注释
RET		；从一个过程返回
INC	BX	；BX 寄存器加 1
ADD	CX, 25	；将 25 加到 CX 寄存器中

4.2.5 伪操作

汇编语言支持许多语句，那些语句允许你控制源程序汇编与列表的方式。这些语句称为**伪操作**，它们只在程序汇编过程中起作用，并且产生机器不可执行的代码。基本的伪操作在下面几节加以说明。第 25 章将详细介绍所有的伪操作，任何时候都可以作为参考材料使用。

1. PAGE 和 TITLE 列表伪操作

PAGE 和 **TITLE** 伪操作用来控制被汇编的程序的列表格式。这是它们仅有的用途，并且不影响其后程序的执行。

在程序的起点，**PAGE** 伪操作指定列在一页上的最大行数和在一行中的最大字符数。它的格式是：

PAGE [length] [, width]

例如，伪操作 **PAGE 60, 132**，每页长度是 60 行，每行宽度是 132 个字符。

典型的汇编程序，每页的行数可以在 10 到 255 的范围内，而每行的字符数可以在 60 到 132 的范围内。省略 **PAGE** 语句，汇编程序默认为 **PAGE 50, 80**。

假定程序定义 **PAGE** 的最大行数为 60。当汇编程序打印所汇编的程序并已经列出 60 行时，会自动地翻到下一页的顶端并把页数加 1。

你可能还想把一页从程序列表的一个指定行强行退出来，比如段结束。在所要求的行上，简单写上没有操作数的 **PAGE** 就可以了。遇到 **PAGE** 时，汇编程序翻到下一页的顶端，从那里继续进行列表。

可以使用 TITLE 伪操作为程序产生一个标题，打印在程序列表的每页的第 2 行。只可以在程序的开始，编写一次 TITLE。它的格式是：

```
TITLE text [comment]
```

对于文本(text)，通常的做法是使用程序的名字作为磁盘上的目录。例如，把程序命名为 ASMSORT，编写此名字并在其后加上可选的说明性注释(不要求用“;”作为前导)，整个长度可达到 60 个字符，如下所示：

```
TITLE ASMSORT Assembly program to sort CD title
```

2. 段(SEGMENT)伪操作

正如在第 2 章所讨论的，.EXE 格式的汇编程序是由一个或多个段组成的。在实模式下，堆栈段定义堆栈存储区，数据段定义数据项，而代码段则提供可执行的代码。定义段的伪操作 SEGMENT 和 ENDS 的格式如下：

名字	操作	操作数
segment-name	SEGMENT	[align] [combine] ['class']
...		
segment-name	ENDS	

SEGMENT 语句定义段的开始。段名(segment-name)必须存在，必须是唯一的，而且必须遵循汇编语言命名的惯例。ENDS 语句指明段的结束并包含与 SEGMENT 语句相同的名字。在实模式下，段的大小最大是 64KB。SEGMENT 语句的操作数可包括 3 种类型的选项：定位，组合，以及类别：

- 定位(align)选项指明段开始的边界。典型的要求是 PARA，它使段定位在小段的边界，这样一来，起始地址正好能被 16 或 10H 整除。省略定位操作符，汇编程序默认为 PARA。
- 组合(combine)选项指明：当汇编后进行连接时，本段是否要与其他段组合在一起(稍后在“连接程序”中解释)。组合类型是 STACK，COMMON，PUBLIC，以及 AT expression(AT 表达式)。例如，堆栈段通常定义为：

```
segment-name SEGMENT PARA STACK
```
- 当你打算把分别汇编的程序在连接时组合在一起的时候，可以使用 PUBLIC 和 COMMON。否则，当程序不要和其他程序组合时，可以省略这个选项或写上 NONE。
- 类别(class)选项包在撇号(“'”)中，它在连接时用于把相关的段组成一组。本书使用的类别是代码段用 'code' (Microsoft 推荐)，数据段用 'data'，而堆栈段用 'stack'。

图 4-1 中的不完整的程序说明有各种选项的 SEGMENT 语句。注意：程序用定位(PARA)、组合(STACK)和类别('Stack')等类型定义堆栈段。

```

        page      60,132
TITLE   A04ASM1  Segments for an .EXE Program
;-----
STACK   SEGMENT PARA STACK 'Stack'
        ...
STACK   ENDS
;-----
DATASEG SEGMENT PARA 'Data'
        ...
DATASEG ENDS
;-----
CODESEG SEGMENT PARA 'Code'
        MAIN     PROC   FAR
        ...
MAIN     ENDP      ; 过程结束
CODESEG ENDS      ; 段结束
        END      MAIN ; 程序结束

```

图 4-1 为 .EXE 程序定义段

3. PROC 伪操作

代码段包含程序的可执行代码,该程序是由一个或多个过程组成的,过程的开始用 PROC 伪操作定义,结束则是用 ENDP 伪操作定义。格式如下:

名字	操作	操作数	注释
procedure-name	PROC	FAR	: 过程开始
...			
procedure-name	ENDP		: 过程结束

过程名(procedure-name)必须存在,必须是唯一的,而且必须遵循汇编语言的命名惯例。在这里的操作数是 FAR,它与程序的执行有关。当请求执行一个程序时,装入程序使用这一过程作为入口点,去执行它的第一条指令。

ENDP 伪操作指明过程的结束并包含和 PROC 语句相同的名字,使汇编程序能把结束与开始联系起来。由于一个过程必须完全包含在一个段内,ENDP 定义过程的结束要在 ENDS 定义段结束以前,如图 4-1 所示。

代码段可以包含任意数量的过程作为子程序,每个过程有它自己的一组相匹配的 PROC 和 ENDP 语句。附加的 PROC 通常使用(或默认为)NEAR 作为操作数,这会在第 7 章说明。

4. END 伪操作

如前所述,ENDS 伪操作结束一个段,而 ENDP 伪操作则是结束一个过程。END 伪操作结束一个完整的程序并作为最后语句出现,如图 4-1 所示。它的格式(操作和操作数)是:

```
END [procedure-      name]
```

如果程序不执行,操作数可以是空白的。例如,你可能打算只是汇编数据定义,或者想把程序与另一模块相连接。在大多数程序中,操作数包含第一个名字或者是 PROC 指定为 FAR 的过程名,程序要从这里开始执行。

5. ASSUME 伪操作

.EXE 程序使用 SS 寄存器访问堆栈, DS 访问数据段, CS 访问代码段。为此, 必须告诉汇编程序在程序当中每个段的用途。所需要的伪操作是 ASSUME, 在代码段中的编码如下:

```
ASSUME    SS:stackname, DS:dasegname, CS:codegname, ...
```

SS: stackname 的意思是让汇编程序把堆栈段的名称与 SS 寄存器联系在一起, 对于所示的其他操作数都是类似的。操作数可以按照任何次序出现。ASSUME 还可以包含 ES 寄存器的入口, 如 ES:dasegname。假如你的程序不使用 ES, 则可以省略对 ES 的说明或编码为 ES: NOTHING。(从 MASM 6.0 开始, 汇编程序自动为代码段生成 ASSUME。)

如同其他伪操作一样, ASSUME 只是一个帮助汇编程序把符号码转换成机器码的信息, 你仍然必须编指令使程序在开始执行时实实在在地把地址装入到段寄存器中。

6. 处理器伪操作

大多数汇编程序都假定源程序是运行在基本的 8086 级别的计算机上。结果是当你用了由后来的处理器所提供的指令或特性时, 必须用处理器伪操作来通知汇编程序, 比如 .286, .386, .486, 或 .586。该伪操作可以直接出现在指令之前, 代码段之前, 或在保护模式下甚至可以出现在源程序的开始。还有, 使用扩充寄存器, 如 EAX, 需要使用 .386 伪操作。下面是要求处理器伪操作的某些指令:

.286	.386	.486
POPA	MOVSB/MOVB	CMPXCHG
PUSHA	SHLD/SHRD	XADD

4.3 常规的段伪操作

可执行程序的两基本类型是 .EXE 和 .COM。我们首先阐述有关 .EXE 程序的要求, 留下 .COM 程序到第 5 章说明。图 4-2 提供展示堆栈、数据和代码段的 .EXE 程序的框架。以下按行号来解释程序中的语句:

行	解释
1	使用 PAGE 伪操作确定每页为 60 行与 132 列。
2	使用 TITLE 伪操作指出程序名为 A04ASM1。
3	使用在第 3、7 和 11 行的注释清楚地表明所定义的 3 个段的开始。
4-6	定义堆栈段 STACK (但在本例中没有它的内容)。
8-10	定义数据段 DASEG (但在本例中没有它的内容)。
12	定义代码段 CODESEG。
13-20	定义代码段仅有的过程, 在本例中命名为 MAIN。这一过程说明 .EXE 程序所共同要求的初始化和退出。初始化的两个要求是: (1) 通知汇编程序哪些段与段寄存器相关联, (2) 将数据段的地址装入 DS。
14	使用 ASSUME 伪操作告诉汇编程序把段和段寄存器联系起来, 在这里, STACK 和 SS, DASEG

和 DS, CODESEG 和 CS 是相关联的:

```
ASSUME SS:STACK, DS:DATASEG, CS:CODESEG
```

1		page	60,132	
2	TITLE	A04ASM1	Skeleton of an .EXE Program	
3				
4	STACK	SEGMENT	PARA	'Stack'
5		...		
6	STACK	ENDS		
7				
8	DATASEG	SEGMENT	PARA	'Data'
9		...		
10	DATASEG	ENDS		
11				
12	CODESEG	SEGMENT	PARA	'Code'
13	MAIN	PROC	FAR	
14		ASSUME	SS:STACK, DS:DATASEG, CS:CODESEG	
15		MOV	AX, DATASEG	;把数据段地址
16		MOV	DS, AX	;设置在DS中
17		...		
18		MOV	AX, 4C00H	;结束处理
19		INT	21H	
20	MAIN	ENDP		;过程结束
21	CODESEG	ENDS		;段结束
22		END	MAIN	;程序结束

图 4-2 .EXE 程序的框架

利用段与段寄存器的联系, 汇编程序可以确定在堆栈中各项的偏移地址、在数据段中各数据项的偏移地址和在代码段中指令的偏移地址。例如, 在代码段中的每条机器指令具有确定的长度。以机器语言表示的第一条指令应该在偏移地址为 0 处, 假如它是 2 字节长, 那么第二条指令应该位于偏移地址为 2 处, 以此类推。

15, 16 初始化数据段的地址存入 DS 中:

```
MOV AX, DATASEG      ; 取得数据段地址
MOV DS, AX            ; 把地址存入 DS
```

第一条 MOV 指令把数据段地址装入到 AX 寄存器中, 第二条 MOV 则把地址从 AX 复制到 DS。因为处理器只允许从通用寄存器到段寄存器传送数据, 所以这两条 MOV 指令是必需的。语句 MOV DS, DATASEG 是非法的, 因为它试图把数据直接从存储器传送到 DS。第 5 章会对初始化段寄存器作更详细的讨论。

18, 19 要求终止程序的执行并返回到操作系统。下一节会详细说明。

22 使用 END 伪操作告诉汇编程序这是源程序的终点。MAIN 操作数指出名为 MAIN 的过程是随后程序执行的入口点。MAIN 可以是汇编程序能够接受的其他名字, 这只要 PROC、ENDP 与 END 使用相同的名字即可实现。

你所定义的段的顺序通常是不重要的。图 4-2 将它们定义如下:

```
STACK            SEGMENT            PARA            STACK 'Stack'
DATASEG          SEGMENT            PARA            'Data'
CODESEG          SEGMENT            PARA            'Code'
```

注意: 图中的程序是用符号语言编码的。为了执行它, 必须使用汇编程序和连接程序把它们翻译成可执行的机器码, 成为 .EXE 程序。

正如第2章所讨论的,当装入程序为执行一个.EXE程序把它从磁盘读入存储器时,就在可用内存的某个小段边界上构造一个256个字节(100H)的PSP(程序段前缀),并立即把程序存放在紧跟着的边界上。然后,装入程序执行以下操作:

- 初始化代码段地址存入CS中;
- 初始化堆栈地址存入SS中;
- 初始化PSP的地址存入DS与ES中。

装入程序初始化CS:IP和SS:SP寄存器,以便处理器可以分别访问代码与堆栈段。然而,你的程序通常需要在DS中(通常也在ES中)的数据段地址,而不是PSP的地址。因此,必须要用数据段的地址去初始化DS,正如在图4-2中用两条MOV指令所示的那样。

现在,虽然对这种初始化还不算很了解,但可喜的是,实际上每个.EXE程序初始化的步骤都是一样的,这样你只要编写一次,然后每次加以复制就可以了。

4.3.1 结束程序执行

INT 21H是公用的DOS中断,它使用AH寄存器中的功能码指定所要完成的动作。INT 21H的众多功能包括:键盘输入,屏幕处理,磁盘I/O,以及打印输出。这里与我们有关的功能是4CH,INT 21H把它识别成结束程序执行的请求。还可以利用这一操作,为后继的批处理文件测试(通过IF ERRORLEVEL语句)传递AL中的返回码,就像这样:

```
MOV AH, 4CH      ; 请求结束处理
MOV AL, retcode   ; 选择返回码
INT 21H          ; 调用中断服务程序
```

对于正常完成的程序,返回码通常是0。也可以把两条MOV指令重新编写为一个语句(如图4-2所示):

```
MOV AX, 4C00H    ; 请求正常退出
```

4.3.2 源程序举例

图4-3把前述的信息组合成一个简单而又完整的汇编源程序,该程序在AX寄存器中把两个数据项相加。段是用以下方法定义的:

- STACK包括一个项:DW(定义字),它定义32个被初始化为0的字,对于小程序而言大小是足够的。
- DATASEG定义3个字,命名为FLDD(用215初始化),FLDE(用125初始化),以及FLDF(未初始化)。
- CODESEG包含程序的可执行指令,最先的两个语句PROC和ASSUME生成的是不可执行码。

ASSUME伪操作告诉汇编程序完成以下任务:

- 赋值STACK到SS寄存器,使处理器使用SS中的地址去访问STACK。
- 赋值DATASEG到DS寄存器,使处理器使用DS中的地址去访问DATASEG。
- 赋值CODESEG到CS寄存器,使处理器使用CS中的地址去访问CODESEG。

```

page 60,132
TITLE A04ASM1 (EXE) Move and add operations
;-----
STACK SEGMENT PARA STACK 'Stack'
DW 32 DUP(0)
STACK ENDS
;-----
DATASEG SEGMENT PARA 'Data'
FLDD DW 215
FLDE DW 125
FLDF DW ?
DATASEG ENDS
;-----
CODESEG SEGMENT PARA 'Code'
MAIN PROC FAR
ASSUME SS:STACK,DS:DATASEG,CS:CODESEG
MOV AX,DATASG ;把数据段地址
MOV DS,AX ;设置在DS中
MOV AX,FLDD ;把0215传送到AX
ADD AX,FLDE ;把0125加到AX中
MOV FLDF,AX ;在FLDF中存放和
MOV AX,4C00H ;结束处理
INT 21H
MAIN ENDP
CODESEG ENDS
END MAIN ;过程结束
;段结束
;程序结束
```

图 4.3 具有常规段的.EXE 程序

当为了执行而把程序从磁盘装入到存储器时,装入程序在SS和CS中设置正确的段地址,并且正如头上两条MOV指令所表示的那样,程序必须初始化DS(通常还有ES)。
在第5章里,将汇编,连接,以及执行这一程序。

4.4 简化的段伪操作

汇编程序提供了一些定义段的捷径。为了使用它们,在定义任何段之前,必须初始化存储模型。不同的模型告诉汇编程序如何使用段,为目标码提供足够的空间,以及确保最佳的执行速度。它的格式(包括前面的圆点)是:

```
.MODEL memory-model
```

存储模型(memory-model)可以是 Tiny(微型), Small(小型), Medium(中型), Compact(紧凑型), Large(大型), Huge(巨型), 或 Flat(平面型)。根据MASM 6.0和TASM 4.0, Tiny模型是为.COM程序而设计的,该程序的数据、代码和堆栈都在一个64KB的段里。Flat模型为代码和数据定义了一个多达4GB的区域,程序采用32位寻址方式并在保护模式的Windows下运行。其它模型的要求是:

模型	代码段数量	数据段数量
Small	1<=64KB	1<=64KB
Medium	任意数量,任意大小	1<=64KB
Compact	1<=64KB	任意数量,任意大小
Large	任意数量,任意大小	任意数量,任意大小
Huge	任意数量,任意大小	任意数量,任意大小

可以在单独的(即不需和其他程序连接的)程序中使用这些模型中的任何一个。Small

模型适用于本书中大多数例子,汇编程序采用的地址是近的(在 64K 范围内)并产生 16 位的偏移地址。相反,对于 Compact 模型,汇编程序采用 32 位地址,因而也就需要更多的执行时间。Huge 模型和 Large 模型是一样的,但可以包含诸如大于 64K 数组的变量。MODEL 伪操作自动地为所有模型产生所需要的 ASSUME 语句。

定义堆栈、数据和代码段的伪操作的格式(包括前面的圆点)是:

```
.STACK [size]
.DATA
.CODE [segment-name]
```

上述每一个伪操作都会使汇编程序产生所需要的 SEGMENT 语句和与其相匹配的 ENDS。默认的段名(不必定义它)是 STACK, DATA, 以及 TEXT(对于 Tiny、Small、Compact 和 Flat 模型的代码段而言)。DATA 和 TEXT 前面的破折号(或下划线)是必须的。默认的堆栈大小是 1024 个字节,可以不管它而自己定义大小。正如编码格式所指明的那样,可以不管代码段的默认段名,使用这些伪操作去确定 3 个段在程序中的位置。但是要注意:现在用于初始化 DS 中的数据段地址的指令是:

```
MOV AX, @data      ; 用数据段的地址来
MOV DS, AX         ; 初始化 DS
```

图 4-3 给出一个使用常规方式定义段的程序的例子。现在图 4-4 提供同样的例子,但这次是使用简化段伪操作 STACK, DATA, 以及 CODE。存储模型指定为 Small, 位于第 4 行。堆栈定义为 64 字节(32 个字)。注意:汇编程序不产生常规的 SEGMENT 和 ENDS 语句,并且不需要编写 ASSUME 语句。

	page	60,132	
TITLE	A04ASM2 (EXE)	Move and add operations	

	.MODEL	SMALL	
	.STACK	64	; 定义堆栈
	.DATA		; 定义数据
FLDD	DW	215	
FLDE	DW	125	
FLDF	DW	?	

	.CODE		; 定义代码段
MAIN	PROC	FAR	
	MOV	AX, @data	; 把数据段地址
	MOV	DS, AX	; 设置在 DS 中
	MOV	AX, FLDD	; 把 0215 传送到 AX
	ADD	AX, FLDE	; 把 0125 加到 AX 中
	MOV	FLDF, AX	; 在 FLDF 中存放和
	MOV	AX, 4C00H	; 结束处理
	INT	21H	
MAIN	ENDP		; 过程结束
	END	MAIN	; 程序结束

图 4-4 使用简化的段伪操作的 EXE 程序

下一章将会看到,汇编程序对于使用简化段伪操作编码的程序和使用常规段伪操作的程序在处理上是有微小差别的。

.STARTUP 和.EXIT 伪操作

MASM 6.0 引入.STARTUP 和.EXIT 伪操作来简化程序的初始化和结束。.STARTUP 产生指令去初始化各段寄存器，而.EXIT 则产生功能为 4CH 的 INT 21H 指令退出程序。这些伪操作需要.MODEL 伪操作，并且对于除了 Flat 模型外的所有存储模型都是有效的。为了学习汇编语言，本书中的例子使用全套指令系统而把捷径留给更有经验的程序设计人员。

4.5 保护模式下的初始化

程序在 Windows 下以保护模式运行并定义 Flat 型的存储模型。由于段的寻址已扩展到 32 位，所以单个可寻址的程序区可以达到 4GB。程序的框架模型是：

```
.386 or .486                ; 首先是处理器伪操作
.MODEL FLAT, STDCALL
.STACK
.DATA                        ; 随后是所有数据
.CODE                        ; 随后是指令代码
END
```

在.MODEL 语句前编写处理器伪操作使汇编程序采用 32 位寻址方式。STDCALL 告诉汇编程序对于名字和过程调用使用标准的规定。由于不必要把段：偏移值转换成实际地址，所以处理器的操作会更为有效。在实模式下，数据和指令的偏移值是 16 位，而在保护模式下偏移值是 32 位。

使用 DWORD 把段对准在双字地址上，能加速对 32 位数据总线的存储器的存取。类型 USE32 指示汇编程序产生适合于 32 位保护模式的代码：

```
segment-name SEGMENT DWORD USE32
```

由于这些处理器的 DS 仍然是 16 位，所以 DS 寄存器的初始化如下：

```
MOV  EAX, DATASEG          ; 得到数据段的地址
MOV  DS, AX                 ; 把 16 位部分装入 DS
```

STI, CLI, IN, 以及 OUT 指令在实模式下是可用的，但不允许用在保护模式下。

4.6 定义数据类型

如前所述，.EXE 程序中的数据段包含常数，工作区，以及输入/输出区。汇编程序提供了一组伪操作，它们允许定义不同类型和长度的项，例如，DB 定义字节，而 DW 定义字。数据项可以包含未定义的(即未初始化的)值，也可以包含初始化的常数，所定义的或者是字符串，或者是数字值。下面是数据定义格式：

[name]	Dn	expression
--------	----	------------

(1) 名字(Name)。程序是通过名字来访问数据项的。上述定义中的方括号表明名字是可选项。先前的“语句”一节提供了命名规则。

(2) 伪操作(Dn)。定义数据项的伪操作是 DB(字节), DW(字), DD(双字), DF(远字), DQ(4 字), 以及 DT(10 字节), 每一个都明确地指出了所定义的项的长度。MASM 6.0 引入了 BYTE, WORD, DWORD, FWORD, QWORD, 以及 TWORD, 它们分别对应于上述伪操作。原来的术语在汇编语言的许多版本中仍然是通用的。

(3) 表达式(Expression)。操作数中的表达式可以指定未初始化的值或常数值。为了指明未初始化的项, 用问号定义操作数, 比如

```
DATA1 DB ? ; 未初始化项
```

在这种情况下, 当程序开始执行时, DATA1 的初始值是未知的。使用这一项之前的通常做法是向它传送某些值, 这些值必须符合所定义的大小。

可以用操作数去定义常数, 比如

```
DATA1 DB 25 ; 初始化项
```

在整个程序中, 可以自由地使用这个初始化值 25, 甚至可以改变该值。

表达式可以包含用逗号分开的多个常数, 它只受行长度的限制, 如下所示:

```
DATA1 DB 21, 22, 23, 24, 25, 26, ...
```

汇编程序是以相邻的字节从左到右定义这些常数的。对于 DATA1 的访问是访问第一个一字节常数 21(可以把第一个字节看成是 DATA1+0), 而对 DATA1+1 的访问则是访问第二个常数 22。例如, 指令

```
MOV AL, DATA1+3
```

把 24(18H)这个值装入到 AL 寄存器。表达式还允许在如下格式的语句中重复常数:

[name]	Dn	repeat-count DUP(expression)...
--------	----	---------------------------------

以下的例子说明这种重复:

```
DW 10 DUP(?) ; 10 个字, 未初始化
```

```
DB 5 DUP(12) ; 5 个字节, 为 hex 0C0C0C0C0C
```

```
DB 3 DUP(5 DUP(4)) ; 15 个 4
```

第三个例子产生数字 4 的 5 次复制(44444)并重复该值 3 次, 总共是 15 个 4。

表达式可以定义和初始化字符串或数字常数。

4.6.1 字符串

字符串用于描述比如人名或产品说明等。串是在单引号内定义的, 如 'PC', 或在双引号内定义, 如 "PC"。汇编程序把引号中的内容以 ASCII 形式作为目标码存储, 不包括撇号。

在 MASM 中, DB(或 BYTE)是定义 2 个以上字符的字符串的仅有格式, 这些字符以左相邻的方式并按正常的从左到右的顺序(就像名字和地址一样)存放。因此, 对于定义任意长度的字符数据来说, DB 是常用的格式。例如:

```
DB 'Computer City'
```

如果串中必须包含单引号或双引号，可以用以下方法之一来定义它：

```
DB " Crazy Sam's CD Emporium"      ; 双引号用于串，单引号用作撇号
DB ' Crazy Sam"s CD Emporium'       ; 单引号用于串，2 个单引号用作撇号
```

4.6.2 数字常数

数字常数用于定义算术运算的值与存储器地址。常数不定义在引号内，但要跟随一个可选的**基数区分符**，比如十六进制值 12H 中的 H。对于大多数数据定义伪操作，汇编程序把所产生的字节按相反顺序(即从右到左)存放在目标码中。以下是各种数字格式：

(1) **二进制**。二进制格式使用二进制数字 0 和 1，后跟基数区分符 B。二进制格式通常用在位处理指令 AND, OR, XOR, 以及 TEST 中作为识别位的值。

(2) **十进制**。十进制格式使用十进制数字 0 到 9，后跟(可以选择)基数区分符 D，如 125 或 125D。尽管汇编程序为编码的方便允许你用十进制格式定义值，但还是要将十进制值转换成二进制目标码并用十六进制格式表示它们。例如，定义的十进制的 125 就变成了 hex 7D。

(3) **十六进制**。十六进制格式使用十六进制数字 0 到 F，后跟基数区分符 H。由于汇编程序希望对于字母开头的访问是符号名，所以十六进制常数的第一个数字必须是 0 到 9。例如，3DH 和 0DE8H，汇编程序分别把它们存放为 3D 和按字节相反顺序的 E80D。

因为汇编程序要把全部数字值都转换成二进制(并把它表示成十六进制)，十进制 12、十六进制 C 和二进制的 1100 的定义都产生相同的值：二进制的 00001100 或十六进制的 0C，这取决于你如何观察字节的内容。

由于字母 D 和 B 既作为基数区分符又是十六进制数字，这可能会造成某些混乱。作为解决办法，MASM 6.0 引入了 T(代表 10)和 Y(代表二进制)分别用作十进制与二进制的基数区分符。

(4) **实数**。汇编程序把给定的实数(十进制或十六进制常数后跟基数区分符 R)转换成用于数值协处理器的浮点格式。

务必要把字符和数字常数的用法区分开来。例如，定义成 DB '24' 的字符常数会产生两个 ASCII 字符，表示为 hex 3234。而定义成 DB 24 的数字常数则产生二进制数，表示为 hex 18。

4.6.3 定义数据的伪操作

下面列出了用于定义数据的一般伪操作与由 MASM 6.0 所引入的伪操作：

定义	一般的伪操作	MASM 6.0 伪操作
字节	DB	BYTE
字	DW	WORD
双字	DD	DWORD
远字	DF	FWORD
4 字	DQ	QWORD
10 字节	DT	TBYTE

本书使用的是—般的伪操作，因为它们通用。图 4-5 的汇编语言程序中提供 DB、DW、

DD 和 DQ 伪操作定义字符串与数字常数的例子。生成的目标码列在左边。注意未定义值的目标码表现为十六进制的零。这个程序只由一个数据段组成，而没有可执行的指令，它是不适于执行的。

(1) **DB 或 BYTE**: 定义字节。DB 或 BYTE 数字表达式可以定义一个或多个一个字节的数据，每个由 2 个十六进制数字组成。对于无符号数字数据，值的范围是 0 到 255；对于带符号的数据，值的范围是 -128 到 +127。汇编程序把数字常数转换成二进制目标码(以十六进制表示)。

在图 4-5 中，第一个定义 BYTE1 使用 '?' 去指定未初始化的值。DB 数字常数是 BYTE2、BYTE3、BYTE4 和 BYTE5。例如，汇编程序把所定义的值 48 转换成 hex 30。

DB 字符表达式可以包含任意长的串直到行结束。例如，在图中所见的 BYTE6 和 BYTE7。十六进制目标码表示每个字节的 ASCII 字符按正常从左到右顺序列出，其中 20H 代表空格。

BYTE8 表示适用于定义表格的数字与串常数的混合。

```

                                TITLE      page 60,132
                                A04DEFIN (EXE) Define data directives
                                .MODEL     SMALL
                                .DATA
                                ;          DB - Define Bytes:
                                ;          -----
0000 00                          BYTE1    DB      ?           ; 未初始化
0001 30                          BYTE2    DB      48           ; 十进制常数
0002 30                          BYTE3    DB      30H          ; 十六进制常数
0003 7A                          BYTE4    DB      01111010B     ; 二进制常数
0004 000A[ 00 ]                 BYTE5    DB      10 DUP(0)      ; 10个零
000E 50 43 20 45 6D 70          BYTE6    DB      'PC Emporium' ; 字符串
                                ;          ; 数字作为字符
0019 31 32 33 34 35            BYTE7    DB      '12345'
001E 01 4A 61 6E 02 46          BYTE8    DB      01,'Jan',02,'Feb',03,'Mar'
                                ;          ; 月份表
                                ;          ; 65 62 03 4D 61 72
                                ;          -----
                                ;          DW - Define Words:
                                ;          -----
002A FFF0                       WORD1    DW      0FFFF0H       ; 十六进制常数
002C 007A                       WORD2    DW      01111010B     ; 二进制常数
002E 001E R                     WORD3    DW      BYTE8         ; 地址常数
0030 0002 0004 0006 0007        WORD4    DW      2,4,6,7,9     ; 5个常数的表
                                ;          ; 0009
003A 0008[ 0000 ]              WORD5    DW      8 DUP(0)       ; 8个零
                                ;          -----
                                ;          DD - Define Doublewords:
                                ;          -----
004A 00000000                  DWORD1    DD      ?           ; 未初始化
004E 0000A25A                  DWORD2    DD      41562         ; 十进制值
0052 00000018 00000030          DWORD3    DD      24, 48       ; 2个常数
005A 00000001                  DWORD4    DD      BYTE3 - BYTE2 ; 地址差值
                                ;          -----
                                ;          DQ - Define Quadwords:
                                ;          -----
005E 0000000000000000          QWORD1    DQ      0           ; 零常数
0066 395E000000000000          QWORD2    DQ      05E39H       ; 十六进制常数
006E 5AA2000000000000          QWORD3    DQ      41562         ; 十进制常数
                                END

```

图 4-5 字符与数字数据的定义

(2) **DW 或 WORD**: 定义字。DW 或 WORD 伪操作定义长度为一个字(两个字节)的项。DW 数字表达式可以定义一个或多个一个字的常数。对于无符号的数字数据，值的范围是 0 到 65535；对于带符号的数据，值的范围是 -32768 到 +32767。

汇编程序把 DW 数字常数转换成二进制目标码(以十六进制表示)，但是以相反顺序存放

字节。因此,被定义为 12345 的十进制值转换成了 hex 3039,但按 3930 存放。

在图 4-5 中,WORD1 和 WORD2 定义 DW 数字常数。WORD3 定义操作数为一个地址,在这种情况下,是 BYTE8 的偏移地址。产生的目标码是 001E(右边的 R 是指浮动的),而核对一下图中的 BYTE8 的偏移地址(最左边一列)确实就是 001E。

WORD4 定义一个有 5 个数字常数的表。注意:每个常数的长度是一个字(两个字节)。WORD5 定义一个用 8 个 0 初始化的表。

在 MASM 下 DW 字符表达式被限制为 2 个字符,所以仅限于定义字符串。

(3) **DD 或 DWORD:** 定义双字。DD 或 DWORD 伪操作定义长度为双字(4 个字节)的项。DD 数字表达式可以定义一个或多个常数,每个最多 4 个字节(8 个十六进制数字)。对于无符号数字数据,值的范围是 0 到 4294967295;对于带符号的数据,值的范围是 -2147483648 到+2147483647。

汇编程序把 DD 数字常数转换成二进制目标码(以十六进制表示),但是以相反顺序存放字节。因此,汇编程序把定义为 12345678 的十进制数转换成 00BC614EH 并存放成 4E61BC00H。

在图 4-5 中,DWORD2 定义一个 DD 数字常数,而 DWORD3 则定义 2 个数字常数。DWORD4 产生 2 个所定义地址之间的差值,在这种情况下,结果是 BYTE2 的长度,即一个字节。

在 MASM 下 DD 字符表达式被限制为 2 个字符,同这种情况下的 DW 一样没有多大价值。在 4 字节的双字中,汇编程序向右对准字符。

(4) **DQ 或 QWORD:** 定义 4 字。DQ 或 QWORD 伪操作定义长度为 4 个字(8 个字节)的项。DQ 数字表达式可以定义一个或多个常数,每个最多 8 个字节(16 个十六进制数字)。最大的正的 4 字十六进制数是 7 后面跟着 15 个 F。作为这个数大小的一种表示,十六进制 1 后面跟着 15 个 0 等于十进制数 1152921504606846976。

汇编程序处理 DQ 数字值和字符串与它处理 DD 和 DW 数字值是一样的。在图 4-5 中,QWORD1、QWORD2 和 QWORD3 说明这些数字值。

4.6.4 显示数据段

把源程序转换成机器码需要两个步骤:汇编和连接。虽然对于图 4-5,汇编程序没有产生出错信息,但连接程序显示了“Warning: No STACK Segment”(警告:没有堆栈段)和“There were 1 error detected”(发现一个错误)。不管该警告,仍可以用 DEBUG 去查看目标码,如图 4-6 所示。

-d ds:100		
0D98:0100	00 30 30 7A 00 00 00 00-00 00 00 00 00 00 50 43	.00z.....PC
0D98:0110	20 45 6D 70 6F 72 69 75-6D 31 32 33 34 35 01 4A	.Emporium12345.J
0D98:0120	61 6E 02 46 65 62 03 4D-61 72 F0 FF 7A 00 1E 00	an.Feb.Mar...z...
0D98:0130	02 00 04 00 06 00 07 00-09 00 00 00 00 00 00 00Z.
0D98:0140	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 000.....
0D98:0150	00 00 18 00 00 00 30 00-00 00 01 00 00 00 00 009^.....Z.
0D98:0160	00 00 00 00 00 00 39 5E-00 00 00 00 00 00 00 00 00".B..B..
0D98:0170	00 00 00 00 00 00 22 DE-42 D9 21 DE 42 D9 21 DE	
<div style="display: flex; justify-content: space-between; align-items: center;"> ← hexadecimal representation → ← ASCII → </div>		

图 4-6 显示数据段

如果提前看一下第5章内容,那么就可以汇编和连接该程序。然后使用DEBUG装入.EXE文件并键入D DS:100显示数据。右边显示ASCII表达式,如“PC Emporium”。而在左边的十六进制值则指明实际的存储内容。显示的应该和图4-6中偏移值为0100到019D的情况相同。可以预料段地址(在图中是0D98)和程序结束后的数据会不一样。

为了显示而发送DS:100的理由是因为装入程序用PSP的地址设置DS,而这个程序的数据段是从该地址之后100个字节开始的。以后,为.EXE程序使用DEBUG时,因为该程序把DS初始化为数据段地址,所以可以用DS:0去显示它。

4.7 相等伪操作

汇编语言提供了等号,EQU和TEXTEQU伪操作。它们是为了用其他名字再定义符号名,以及用名字再定义数字值而提供的。这些伪操作不能把数据存储起来,也就是说,在程序执行时,是不能在EQU项做加法的,而是汇编程序在其他语句中用所定义的值去代替。

相等伪操作的优点是许多语句都可以使用所赋予的值。如果这个值必须要修改时,只需要修改相等语句。它的效果是使程序更具可读性,而且更易于维护。

(1) 等号伪操作。等号伪操作允许把表达式的值指派给一个名字,并且可以在程序中指派任意次。以下例子说明它的用法:

```
VALUE_OF_PT=3.1416
RIGHT_COL=79
SCREEN_POSITIONS=80*25
```

使用上述伪操作的例子是:

```
IMUL AX, VALUE_OF_PT      ; AX 乘以 3.1416
CMP  BL, RIGHT_COL        ; 把 BL 与 79 比较
MOV  CX, SCREEN_POSITIONS ; 把 2000 传送到 CX
```

当用这一伪操作定义双字值时,首先要使用386伪操作去通知汇编程序:

```
.386
DQWORD1=42A3B05CH
```

(2) EQU伪操作。考虑以下在数据段中编写的EQU语句:

```
FACTOR EQU 12
```

在这种情况下,名字FACTOR可以是汇编程序能够接受的任何名字。现在,每当FACTOR出现在一条指令中或另一个伪操作中时,汇编程序都会用12这个值来替代。例如,汇编程序把伪操作

```
TABLEX DB FACTOR DUP(?)
```

转换成它的等效值

```
TABLEX DB 12 DUP(?)
```

TABLEX只能用EQU定义一次,它不能用另一个EQU来再定义。指令也可以包含相等的操作数,如下所示:

```
RIGHT_COL EQU 79
...
MOV CX, RIGHT_COL      ; 把 79 传送到 CX
```

还可以使符号名相等，比如在以下的代码中：

```
ANNL_TEMP DW 0
...
AT      EQU ANNL_TEMP
MPY     EQU MUL
```

第一个 EQU 使 AT 与所定义的项 ANNL_TEMP 相等。对于任何包含操作数 AT 的指令，汇编程序都会用 ANNL_TEMP 的地址去取代它。第二个 EQU 能使程序用 MPY 去代替规则的符号指令 MUL。

(3) **TEXT EQU 伪操作**。MASM 6.0 引入 TEXT EQU 伪操作来再定义文本数据，格式如下：

name	TEXT EQU	<text>
------	----------	--------

可以在角括号(<>)内定义字符文本(text)，例如：

```
PROMPT_MSSGE TEXT EQU <' Add, Change, or Delete?' >
```

使用方法如下：

```
USER_PROMPT DB PROMPT_MSSGE
```

汇编程序把这个定义转换为

```
USER_PROMPT DB ' Add, Change, or Delete?'
```

4.8 要 点

- 在一行中，注释的前面要用分号。
- 在汇编语言中，保留字只为达到专门目的而使用，即它是专用的。
- 标识符是程序中适用于项的名字。标识符的两种类型是名字(指的是数据项的地址)和标号(指的是指令的地址)。
- 操作通常是用于定义数据区和编码的指令。操作数提供操作作用于其上的信息。
- 程序由一个或多个段组成，每个段都是从小段边界开始的。
- ENDS 伪操作结束一个段，ENDP 结束一个过程，而 END 则是结束程序。
- ASSUME 伪操作把段寄存器 CS、DS 与 SS 和与其相应的段名联系起来。
- 对于 .EXE 程序，通常要用数据段的地址去初始化 DS。
- 对于简化段伪操作，在定义任何段之前要初始化存储模型。
- INT 21H 的 4CH 功能是退出程序的标准指令。
- 数据项的命名应当是唯一的和具有描述性的。
- DB(或 BYTE)是定义字符串的优先选用格式，因为它允许串的长度大于 2 个字节并把它们转换成正常的从左到右的序列。
- 十进制与二进制(十六进制)常数会产生不同的值。考虑比较一下十进制 25 和十六进制 25 在加法中的效果：

```
ADD CX, 25      ; 加 25
```

ADD CX, 25H ; 加 37

- 对于 DW、DD 和 DQ，汇编程序在目标码里按相反的字节顺序存放数字值。
- DB 项用于处理半个寄存器(AL, BL, 等等), DW 用于整个寄存器(AX, BX, 等等), 而 DD 则用于扩充寄存器(EAX, EBX, 等等)。较长的数据项需要特殊的处理。

4.9 习 题

- 4-1. 编译程序与汇编程序的区别。
- 4-2. 汇编语言中的保留字是什么? 给出 2 个例子。
- 4-3. 汇编语言中标识符的两种类型是什么?
- 4-4. 对于以下在数据段中所定义的项, 确定哪些是合法的, 如果不合法, 请说明原因。
(a) \$50, (b) AT&T, (c) \$_A, (d) 23AC, (e) EBX。
- 4-5. 说明指令与伪操作之间的区别, 每种各举 2 个例子。
- 4-6. 给出伪操作, 使汇编程序在程序列表时(a)在页的顶端打印一个标题, (b)前进到新页。
- 4-7. 用什么方法(如果有的话)使汇编程序把大小写字母区别看待?
- 4-8. SEGMENT 伪操作的格式是
Name SEGMENT align combine 'class'
说明(a)align(定位), (b)combine(组合), (c) 'class' (类别)的用途。
- 4-9. (a)说明过程的意义。(b)什么时候把过程定义为 NEAR? (c)什么时候把过程定义为 FAR? (d)如何定义过程的开始与结束?
- 4-10. 与结束(a)一个过程, (b)一个段, (c)一个程序相关的语句是什么?
- 4-11. 指出结束汇编的语句和结束执行的语句。
- 4-12. 给出堆栈段、数据段和代码段的名称分别为 STKSEG、DATSEG 和 CODSEG, 写出所需要的 ASSUME 语句。
- 4-13. 考虑与 INT 21H 一起使用的指令 MOV AX, 4C00H。(a)说明指令完成什么工作, (b)说明 4C00H 的用途。
- 4-14. 为了简化段伪操作, .MODEL 伪操作提供了 Tiny、Small、Medium、Compact、Large 和 Flat 型的存储模型。每种模型在什么情况下使用?
- 4-15. 给出以下数据伪操作所产生的字节长度: (a)DW, (b)DQ, (c)DB, (d)DD。
- 4-16. 定义名为 CO_NAME 的字符串, 其中包含 "Internet Services" 作为常数。
- 4-17. 在名为 ITEM1 到 ITEM5 的数据项中, 定义以下数字值, 分别为:
(a) 一个 1 字节项包含等价于十进制 71 的十六进制值。
(b) 一个 2 字节项包含一个未定义值。
(c) 一个 4 字节项包含等价于十进制 7524 的十六进制值。
(d) 一个 1 字节项包含等价于十进制 47 的二进制值。
(e) 一个项包含连续的字的值是 6, 9, 14, 18, 23, 29, 31 和 38。
- 4-18. 写出(a)DB 82, (b)DB '82', (c)DB 4 DUP('5')所产生的十六进制目标码。

4-19. 对于(a)DB 72, (b)DW 2ABEH, (c)DD 1EB6C3H, (d)DQ 24C3E29 确定由汇编程序存放的十六进制目标码。

4-20. (a)使用 EQU 伪操作把 16H 的值再定义为 ATTRIBUTE。(b)用 MOV 指令把 ATTRIBUTE 传送到 BL 寄存器。

目的：阐明汇编、连接与执行汇编语言程序的各个步骤

5.1 引言

本章阐述键入汇编语言程序并对它进行汇编、连接与执行的过程。用汇编语言进行编码的符号指令称为源程序。用汇编程序把源程序翻译成机器码，称为目标程序。最后，再用连接程序完成对于目标程序的机器寻址，生成可执行模块。

有关汇编的几节，说明如何请求汇编程序的执行，汇编程序提供诊断(包括出错信息)并产生目标程序。还要详细说明汇编程序列表并大致地介绍汇编程序如何处理源程序。

有关连接的几节，说明如何请求连接程序的执行，以便能生成可执行模块。还要说明产生连接映像的细节，以及连接程序的诊断。最后讨论如何请求可执行模块的执行。

最后一节，解释如何去编码、转换和执行.COM程序。

5.2 为汇编与执行准备程序

在第4章里，图4-3只表示了程序的源码，它还不是可执行的格式。为了键入这个程序，可以使用任何一种编辑程序或字处理程序，产生标准的、未格式化的ASCII文件。通过把程序和文件装入到磁盘中，可获得很高的效率。调出编辑程序，键入在图4-3中的程序语句并把结果文件命名为A05ASM1.ASM。

尽管空格对于汇编程序并不重要，但是如果能使名字、操作、操作数和注释始终保持列的对齐，会增强程序的可读性。许多编辑程序为了有助于列的对齐，tab键每8个位置会停一次。

一旦为该程序键入了全部语句，就应该检查代码的准确性。按现在的情况，该源程序只是一个不能执行的文本文件，必须首先对它进行汇编和连接。图5-1提供了一张汇编、连接与执行程序所要求的步骤图。

1. 汇编步骤包括把源码翻译成目标码并产生中间的.OBJ(目标)文件或模块(在前面的章节里，已经见过机器码和源码的例子)。汇编程序的任务之一就是为数据段中的每个数据项和代码段中的每条指令计算偏移值。汇编程序还要在所产生的.OBJ模块的前面直接建立一个首

部(header), 首部的一部分包含不确定的地址信息。.OBJ 模块不是真正的可执行格式。

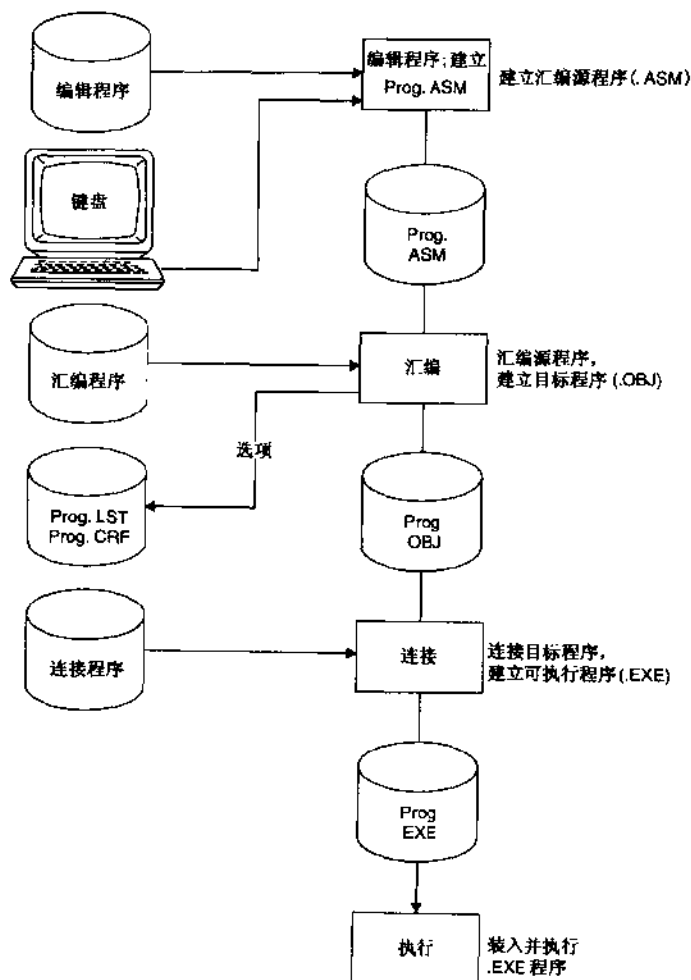


图 5-1 汇编、连接以及执行的步骤

2. 连接步骤是把.OBJ 模块转换成.EXE(可执行的)机器码模块。连接程序的任务包括确定由汇编程序遗留下来的不能确定的地址, 并把分别汇编的程序组合成一个可执行的模块。

3. 最后一步是装入准备执行的程序。装入程序知道程序将要装入到存储器的哪个地方, 所以它能解决首部仍无法确定的任何余下地址。装入程序略去首部并在程序装入存储器之前直接建立程序段前缀(PSP)。

5.2.1 汇编源程序

汇编程序把源语句转换成机器码, 并且在屏幕上显示出错信息。典型的错误包括违反命名规则的名字, 拼写不正确的操作(比如用 MOVE 代替 MOV), 以及操作数含有未定义的名字。由于有许多可能的错误(100 个或更多)和许多不同的汇编程序版本, 所以具体情况可以查阅汇编程序手册。汇编程序试图校正某些错误, 但在任何情况下, 只要重新装入编辑程序,

校正.ASM 源程序并重新汇编它就可以了。

汇编的步骤中可选的输出文件是目标(.OBJ)，列表(.LST)，以及交叉引用(.CRF 或.SBR)。通常需要.OBJ 文件，这个文件是连接程序形成可执行格式所要求的。往往还需要.LST 文件，尤其是当它包含出错诊断信息或是你想检查一下所生成的机器码的时候。对于大型程序，.CRF 文件可能是有用的，在这里，你可以看到哪些指令引用了哪些数据项。除此之外，.CRF 文件使汇编程序产生.LST 文件中项的语句编号，而它是.CRF 文件所引用的。后面几节会详细说明这些文件。

有关汇编与连接方面的详细内容可以参阅附录 E。

5.2.2 使用常规的段定义

图 5-2 提供了汇编程序所产生的列表，名为 A05ASM1.LST。行的宽度是 PAGE 的项目所指定的 132 个位置。

注意，在列表的顶部汇编程序是如何按 PAGE 和 TITLE 伪操作的规定工作的。包括 SEGMENT, PROC, ASSUME 和 END 在内没有一个伪操作生成机器码，因为它们仅仅是向汇编程序提供信息。列表是按以下几部分水平排列的：

1. 最左端列出每行的行号。
2. 第二部分表示数据项和指令的十六进制偏移地址。
3. 第三部分表示所产生的十六进制格式的机器码。
4. 靠右边部分是原始的源代码。

程序本身由垂直的 3 段组成，每段都带着它自己的数据或指令的偏移值。每段包含一个 SEGMENT 伪操作，它通知汇编程序把段对准在某个地址上，该地址是能被 hex 10 整除的。SEGMENT 语句本身不产生机器码。装入程序把每段的内容装入存储器并把段地址初始化在段寄存器中，即 STACK 在 SS 中，DATA SEG 在 DS 中，而 CODE SEG 则在 CS 中。段的起点是从该地址起偏移值为零个字节的地方。

1. 堆栈段

堆栈段包含 DW(定义字)伪操作，该操作定义 32 个字，每个字产生一个零值，用(0)标明。这 32 个字的定义是堆栈的实际大小，一个大型程序可能需要许多输入/输出中断，并且要调用很多子程序，这些都会使用堆栈。堆栈段在偏移值 0040H 处结束，0040H 等于十进制值 64(32 字×2 字节)。汇编程序指出所产生的常数为左边的 0020[0000]，即 20H(32)个零字。

如果堆栈对于它包含的所有进栈的项来说太小，那么不论是汇编程序还是连接程序都不会发出警告，而执行的程序可能会在不可预见的情况下失败。

2. 数据段

程序的数据段 DATA SEG 含有 3 个定义的值，全部是 DW(定义字)格式：

- (1) FLDD 定义一个用十进制值 215 初始化的字，汇编程序把 215 翻译成 00D7H(表示在左边)。

A05ASM1 (EXE) Move and add operations		Page 1-1	
1		page	60,132
2		TITLE	A05ASM1 (EXE) Move and add operations
3		-----	
4	0000	STACK	SEGMENT PARA STACK 'Stack'
5	0000	DW	32 DUP(0)
6	0020[
7	0000]		
8			
9	0040	STACK	ENDS
10		-----	
11	0000	DATASEG	SEGMENT PARA 'Data'
12	0000	FLDD	DW 215
13	0002	FLDE	DW 125
14	0004	FLDF	DW ?
15	0006	DATASEG	ENDS
16		-----	
17	0000	CODESEG	SEGMENT PARA 'Code'
18	0000	MAIN	PROC FAR
19		ASSUME	SS:STACK, DS:DATASEG, CS:CODESEG
20	0000	B8 ---- R	MOV AX, DATASEG ; 把数据段地址
21	0003	8E D8	MOV DS, AX ; 设置在 DS 中
22			
23	0005	A1 0000 R	MOV AX, FLDD ; 把 0215 传送到 AX
24	0008	03 06 0002 R	ADD AX, FLDE ; 把 0125 加到 AX 中
25	000C	A3 0004 R	MOV FLDF, AX ; 在 FLDF 中存放和
26	000F	B8 4C00	MOV AX, 4C00H ; 结束处理
27	0012	CD 21	INT 21H
28	0014	MAIN	ENDP ; 过程结束
29	0014	CODESEG	ENDS ; 段结束
30		END	MAIN ; 程序结束
Segments and Groups:			
	Name	Length	Align
CODESEG	0014	PARA
DATASEG	0006	PARA
STACK	0040	PARA
			Combine Class
			NONE 'CODE'
			NONE 'DATA'
			STACK 'STACK'
Symbols:			
	Name	Type	Value
MAIN	F PROC	0000
			CODESEG Length = 0014
FLDD	L WORD	0000
FLDE	L WORD	0002
FLDF	L WORD	0004
			DATASEG
			DATASEG
			DATASEG
0 Warning Errors			
0 Severe Errors			

图 5-2 使用常规段的汇编了的程序

(2) FLDE 定义一个用十进制值 125 初始化的字, 汇编程序把 125 翻译成 007DH。这 2 个常数实际被存入的值分别是 D700 和 7D00, 可以用 DEBUG 检查。

(3) FLDF 是在操作数中用? 为 DW 定义一个未初始化常数的字。该列表表明这个常数是 0000。

FLDD、FLDE 和 FLDF 的偏移地址分别是 0000、0002 和 0004, 这和它们的字段大小有关。

3. 代码段

程序的代码段 CODESEG 包含程序的可执行代码, 它们全部在一个过程(PROC)中。3 个语句确立数据段的可寻址性:

```

                                ASSUME SS:STACK, DS:DATASEG, CS:CODESEG
0000 B8----R                   MOV AX, DATASEG

```

0003 8E D8 MOV DS, AX

- ASSUME 伪操作将每个段和与它相对应的段寄存器相关联。ASSUME 简单地给汇编程序提供信息，它不为此产生机器码。
- 第一条 MOV 指令把 DATASEG “存入” AX 寄存器中。现在指令不能真正地把段存储在寄存器里——汇编程序识别是对段的访问并假定它的地址。注意左边的机器码：B8---R。4 个短划指的是此刻汇编程序不能确定 DATASEG 的地址。只有当目标程序已被连接并为执行而被装入时，系统才能确定这个地址。由于装入程序可以把程序定位于存储器的任何地方，所以汇编程序要把该地址留下不去解决它并用 R(为了可再定位)来指明这一事实，装入程序用实际地址来取代未确定的地址。
- 第二个 MOV 把 AX 寄存器的内容传送到 DS。由于直接用立即数传送到 DS 的指令是无效的，所以为了初始化它，就需要两条指令。

注意：尽管许多程序员会当作一种标准操作把 ES 寄存器初始化，但该程序并不需要 ES 寄存器。

尽管装入程序在它执行而装入一个程序时，会自动地初始化 SS 和 CS，但如果需要的话，把 DS 和 ES 初始化却是你的责任。

虽然所有这些事情看来似乎过分复杂，可实际上现在你还不必去理解它们。在本书的所有程序中，都使用标准的定义和初始化，你可以为每个程序简单地模仿这种编码。为此，在磁盘上存放一个汇编程序的框架，对你想创建的每个新程序，可在适当的名字下，把这一框架程序复制到文件中并使用编辑程序去完成附加的指令。

初始化 DS 寄存器后的第一条指令是 MOV AX, FLDD，它开始于偏移单元 0005 并产生机器码 A1 0000。在列表中，A1(操作)和 0000(操作数)之间的空格只是为了可读性而加的。下一条指令 ADD AX, FLDE，开始于偏移单元 0008 并产生 4 个字节的机器码。指令 MOVF LDF, AX 把 AX 中的和复制到 FLDF，FLDF 在数据段中的偏移值 0004 处。在这个例子中，机器指令的长度是 2 个、3 个或 4 个字节。

程序中的最后一个语句是 END，它含有操作数 MAIN，该操作数与在偏移值 0000 处的 PROC 名字有关。这是在代码段中的一个单元，装入程序在程序开始执行时，要传送控制给这一单元。

下面的程序列表是段与组表，以及符号表。

4. 段与组表

这个表表示了所定义的段和组。注意，段没有按它们编码时同样的序列列出。在这个例子中，汇编程序把它们按照名字的字母表顺序列出(这个程序中不包含组，以后会讨论该内容)。该表提供了每个段的字节长度，定位(全部是小段)，组合类型，以及类别。

5. 符号表

这个表提供在数据段中数据字段的名字(FLDD, FLDE 和 FLDF)和在代码段中指令所用的标号。对于 MAIN(在例子中是唯一的入口)而言，类型 F PROC 指的是远过程(远是因为 MAIN 作为执行的入口点，必须保证在程序之外仍然是已知的)。值的列给出名字、标号与过程的偏移值(从段的起点)。标题为 “Attr”(属性)的列提供段，每个项都是在该段内定义的。

MASM 6.n 还列出了过程, 参数, 以及组表。附录 E 对这些表作了更详细的说明。

5.2.3 使用简化段伪操作

在第 4 章里, 图 4-4 说明了如何使用简化段伪操作编写程序。现在图 5-3 提供的是那个程序的汇编列表。对于简化段伪操作而言, 初始化 DS 如下:

```
MOV AX, @data
MOV DS, AX
```

“Segments and Groups” 下面的符号表的第一部分表示被汇编程序重新命名的 3 个段, 并按字母表顺序列表如下:

- _DATA, 6 字节长。
- STACK, 40H(64)字节长。
- _TEXT, 作为代码段, 14H(20)字节长。

A05ASM2 (EXE)		Move and add operations		Page 1-1	
1			page 60,132		
2		TITLE	A05ASM2 (EXE)	Move and add operations	
3		-----			
4		.MODEL	SMALL		
5		.STACK	64		; 定义堆栈
6		.DATA			; 定义数据
7	0000 00D7	FLDD	DW 215		
8	0002 007D	FLDE	DW 125		
9	0004 0000	FLDF	DW ?		
10		-----			
11		.CODE			; 定义代码段
12	0000	MAIN	PROC FAR		
13	0000 B8 ---- R	MOV	AX,@data		; 把数据段
14	0003 8E D8	MOV	DS,AX		; 的地址设置在 DS 中
15					
16	0005 A1 0000 R	MOV	AX,FLDD		; 把 0215 传送到 AX
17	0008 03 06 0002 R	ADD	AX,FLDE		; 把 0125 加到 AX 中
18	000C A3 0004 R	MOV	FLDF,AX		; 在 FLDF 中存放和
19					
20	000F B8 4C00	MOV	AX,4C00H		; 结束处理
21	0012 CD 21	INT	21H		
22	0014	MAIN	ENDP		; 过程结束
23			END MAIN		; 程序结束

Segments and Groups:					
	N a m e	Length	Align	Combine	Class
DGROUP		GROUP			
_DATA		0006	WORD	PUBLIC	'DATA'
_STACK		0040	PARA	STACK	'STACK'
_TEXT		0014	WORD	PUBLIC	'CODE'

Symbols:					
	N a m e	Type	Value	Attr	
MAIN		F PROC	0000	_TEXT	Length = 0014
FLDD		L WORD	0000	_DATA	
FLDE		L WORD	0002	_DATA	
FLDF		L WORD	0004	_DATA	
@CODE		TEXT		_TEXT	
@FILENAME		TEXT		a05asm2	

0	Warning Errors
0	Severe Errors

图 5-3 使用简化段伪操作的汇编了的程序

标题“Symbols”下所列出的的是在程序中定义的名字或默认名。简化段伪操作提供许多预定义的等同关系，由@符号开始并可以在程序中自由引用。如同@data一样，它们是：

@CODE 等效于代码段的名字_TEXT

@FILENAME 程序名

可以在 ASSUME 和可执行语句中使用@code 和@data，如 MOV AX, @data。

5.3 二遍扫视汇编程序

为了解决在程序中对还未遇到的地址的向前引用问题，典型的汇编程序要 2 遍或多遍从头到尾地扫视源程序。在第一遍扫视期间，汇编程序要读完整的源程序并构造程序所用的名字和标号的符号表。其中 FLDD、FLDE 和 FLDF 的偏移值分别是 0000、0002 和 0004 字节。虽然程序没有定义指令标号，但它们还是会按本身的偏移值出现在代码段中。第一遍扫视确定每条指令所产生的代码数。

在第二遍扫视期间，汇编程序使用第一遍扫视所构造的符号表。因为它已经知道了每个数据字段和指令的长度与相对位置，就能够完成每条指令的目标码。然后就可以根据需要产生各不相同的目标(.OBJ)，列表(.LST)，以及交叉引用(.CRF)文件。

在第一遍扫视中，可能存在的问题是向前引用(forward reference)：在代码段中某些类型的指令可以引用指令的标号，但汇编程序还没有遇到过它的定义。MASM 是根据假设所产生的每条机器指令的长度是多少来产生目标码的。如果第一遍与第二遍扫视关于指令长度有差别，则 MASM 会发出出错信息“两遍扫视间相位错误”。这种错误相对很少，但它一旦出现，就必须跟踪它的成因并加以校正。

从 6.0 版本开始，MASM 可以更有效地处理指令长度，按照需要可进行许多遍对整个文件的扫视。TASM 用一遍扫视去汇编程序，但假如对于向前引用有困难，可以请求它进行一遍以上的扫视。

5.4 连接目标程序

当程序没有出错信息时，下一步就是去连接目标模块 A05ASML.OBJ，它由汇编程序产生并只包含机器码(MASM 6.1 用 ML 命令完成汇编与连接)。连接程序实现以下功能：

- 如果需要可以把一个以上分别汇编的模块组合成一个可执行的程序，比如把 2 个或多个汇编程序组合在一起，或者把一个汇编程序与一个 C 程序组合起来。
- 产生.EXE 模块并用专门的指令把它初始化，以便它为了执行而接着被装入。

一旦把一个或多个.OBJ 模块连接到.EXE 模块中，就可以任意次地执行.EXE 模块。但每当需要修改程序时，必须校正源程序，再次把它汇编成一个.OBJ 模块，然后再把该.OBJ 模块连接成.EXE 模块。即使最初这些步骤并不完全清楚，但你会发现那只不过是小小的体验而已，而这些步骤将变成是自动进行的。

连接步骤输出的文件是可执行文件(.EXE)，映像文件(.MAP)和库文件(.LIB)。参阅附录 E

有关连接程序的细节。

1. 第一个程序的连接映像

对于程序 A05ASM1, 连接程序产生以下的映像:

START	STOP	LENGTH	NAME	CLASS
00000H	0003FH	0040H	STACK	STACK
00040H	00045H	0006H	DATASEG	DATA
00050H	00063H	0014H	CODESEG	CODE
program entry point at 0005:0000				

- 堆栈是第一段并且从程序起点偏移值为 0 处开始。由于定义为 32 个字, 即 64 个字节长, 所以指明了它的长度(40H)。
- 数据段起始于下一个可用小段边界, 偏移值 40H。
- 代码段起始于再下一个小段边界, 偏移值为 50H(某些汇编程序把段重新按字母顺序排列)。
- 程序入口点 0005:0000, 用的是段: 偏移值形式, 指的是第一条可执行指令的相对地址。实际上, 相对起始地址是段位置 5[0], 偏移值 0 字节, 这是与在 50H 的代码段边界相对应的。当装入程序为了执行该程序而把它装入存储器时, 要使用这个值。

在这一阶段, 你很可能碰到的唯一错误是输入了一个不正确的文件名。解决办法是用连接命令重新开始再做。

2. 第二个程序的连接映像

使用简化段伪操作的第二个程序 A05ASM2 的连接映像表明它和上一程序稍有区别的安
排。首先, 汇编程序按字母顺序重新排列各段; 第二, 相继的段是和字(不是小段)边界对齐
的, 如以下连接映像所表示的:

START	STOP	LENGTH	NAME	CLASS
00000H	00013H	0014H	_TEXT	CODE
00014H	00019H	0006H	_DATA	DATA
00020H	0005FH	0040H	STACK	STACK
program entry point at 0000:0000				

- 代码段现在是第一段并起始于距程序起点偏移值为 0 字节处。
- 数据段起始于下一个字边界, 偏移值为 14H。
- 堆栈起始于再下一个字边界, 偏移值为 20H。
- 程序入口点现在是 0000:0000, 它指的是代码段的相对位置起始于段地址为 0, 偏移值也为 0 处。

5.5 执行程序

已经汇编和连接了程序，现在就可以执行它。如果.EXE文件是在默认的驱动器里，可以要求装入程序把这个要执行的文件读到存储器中。为此键入：

A05ASM1.EXE 或 A05ASM1 (没有.EXE扩展名)

如果没有键入文件扩展名，则装入程序会假定它是可执行的.EXE或.COM程序。但是，由于这个程序的输出不可见，建议在DEBUG下面运行它并使用跟踪(T)命令一步步地跟踪它的执行过程。键入如下：

DEBUG n:A05ASM1.EXE

DEBUG装入.EXE程序模块并显示它的短划提示符。

为了观察堆栈段，键入D SS:0。堆栈的内容是全零，因为它就是那样初始化的。

为了观察代码段，键入D CS:0。把显示的机器码和在汇编列表中的代码段加以比较：

B8----8ED8A10000 ...

汇编的列表不能准确地表示出机器码，因为汇编程序不知道第一条指令的操作数地址。现在，可以用查看所显示的代码的办法来确定该地址。

为了观察寄存器的内容，按R键跟着按回车键。SP(堆栈指针)的内容应当是0040H，这是堆栈的大小(32个字=64个字节=40H)。IP(指令指针)应当是0000H。SS和CS为执行而适当地进行了初始化，它们的值取决于程序在存储器中的位置。

准备执行的第一条指令MOV AX, xxxx和后继的MOV指令是有关初始化DS寄存器的。为了执行第一条MOV指令，按T键(为了跟踪)跟着按回车键并注意对IP的影响。为了执行第二条MOV指令，要再一次按T键跟着按回车键。检查DS，它现在是用段地址初始化的。

第三条MOV指令把FLDD的内容装入AX。再次按T键并注意现在AX的内容是00D7。现在按T键去执行ADD指令并注意AX是0154。按T键使MOV把AX存放到数据段的偏移值0004处。

为了检查数据段的内容，键入D DS:0。操作显示3个数据项是D7 00 7D 00 54 01，每个字的字节都是按相反的顺序排列的。

可以使用L去重新装入并重新运行程序或按Q键退出DEBUG对话。

5.6 交叉引用表

汇编程序生成一个可选文件，可以用它产生程序标识符或符号的交叉引用表。文件的扩展名是.SBR(对于MASM 6.1)，.CRF(对于MASM 5.1)，以及.XRF(对于TASM)。然而，仍然必须把该文件转换成一个适当排序的交叉引用文件。有关建立这一文件的细节可参阅

附录 E。

图 5-4 表示的是图 5-2 的程序所产生的交叉引用表。第一列的符号是按字母顺序排列的。第二列的编号表示成 n#, 指明 .LST 文件中的行, 这是每个符号定义所在的行。这一列右边的编号是行号, 指明了该符号是在哪一行上被其他语句引用的。例如, CODESEG 是在第 17 行被定义的, 是在第 19 行和第 29 行被引用的。FLDF 是在第 14 行被定义的, 在第 25+ 行被引用, 其中 “+” 是指在程序执行期间(用 MOV FLDF, AX)修改过它的值。

程序被汇编时产生许多冗余的文件, 可以安全地删除 .OBJ, .CRF 和 .LST 文件。保留 .ASM 源程序以免需要进一步的改变, 还要保留 .EXE 文件以便程序的执行。

Symbol Cross-Reference (# definition, + modification)				
MAIN	18#	28	30	
CODE	17			
CODESEG	17#	19	29	
DATA	11			
DATASEG	11#	15	19	20
FLDD	12#	23		
FLDE	13#	24		
FLDF	14#	25+		
STACK.	4			
STACK	4#	9	19	

图 5-4 交叉引用表

5.7 出错诊断

汇编程序对违反其规则的任何程序错误提供出错诊断。图 5-5 中的程序和图 5-2 中的程序是类似的, 区别在于出于说明性的目的有意插入了几个错误。由于汇编程序版本不同, 诊断会有所区别。这里的错误是:

行	说明
9	FLDF 的定义需要操作数。
14	DX 应当是 DS, 尽管汇编程序不认为这是错误。
16	AS 应当是 AX。
18	FLDQ 应当是 FLDF。
19	字段的大小(字节和字)必须一致(警告)。
22	校正其他错误将使这一诊断消失。
23	MIAN 应当是 MAIN。

出错信息 22 “二遍扫视间相位错”在二遍扫视汇编程序的第一遍扫视时产生的地址与第二遍扫视产生的不同时才会发生。在 MASM 5.1 下面, 为了查出含糊不清的错误要使用 /D 选项列出第一遍与第二遍的文件, 并且比较偏移地址。

```

1
2
3
4
5
6
7 0000 00AF      FLDD      DW      175
8 0002 0096      FLDE      DW      150
9 0004           FLDF      DW
a05asm3.ASM(9): error A2027: Operand expected
10
11
12 0000           .CODE
13 0000      B8 ---- R      MAIN      PROC      FAR
14 0003      8B D0           MOV      AX,@data      ; 数据段地址
15                               MOV      DX,AX      ; 在 DS 中
16                               MOV      AS,FLDD      ; 把0175传送到AX
a05asm3.ASM(16): error A2009: Symbol not defined: AS
17 0005      03 06 0002 R      ADD      AX,FLDE      ; 把0150加到AX中
18 0009      A3 0000 U      MOV      FLDQ,AX      ; 在FLDF中存放和
a05asm3.ASM(18): error A2009: Symbol not defined: FLDQ
19 000C      A2 0000 R      MOV      FLDD,AL      ; 存入字节值
a05asm3.ASM(19): warning A4031: Operand types must match
20 000F      B8 4C00       MOV      AX,4C00H      ; 结束处理
21 0012      CD 21         INT      21H
22 0014           MAIN      ENDP
a05asm3.ASM(22): error A2006: Phase error between passes
23                               END      MIAN
a05asm3.ASM(23): error A2009: Symbol not defined: MIAN

1 Warning Errors
5 Severe Errors

```

图 5-5 汇编程序的出错诊断

5.8 汇编程序位置计数器

汇编程序维护一个位置计数器，它用来在数据段中为每个被定义的数据项作计数统计。图 5-2 和图 5-3 借助于 3 个定义的数据项说明它的作用：

```

0000 ... FLDD DW ...
0002 ... FLDE DW ...
0004 ... FLDF DW ...

```

最初，位置计数器设置为 0，汇编程序在这里建立第一个数据项 FLDD。因为 FLDD 定义成字，所以汇编程序要使位置计数器加 2 成为 0002，并在这里建立 FLDE。因为 FLDE 也是定义成字的，所以汇编程序再一次把它的位置计数器加 2 成为 0004。下一个数据项 FLDF，还是一个字节，位置计数器再一次加 2 成为 0006，但不存在更多的数据项了。

汇编程序提供许多方法去改变位置计数器的当前值。例如，可以使用 EQU 用不同名字再定义数据项(见第 4 章)，使用 ORG 伪操作在特殊的偏移值处开始一个程序(见下一节)，以及使用 EVEN 或 ALIGN 伪操作去使地址对准到偶数边界上(见第 6 章)。

5.9 编写.COM 程序

对于 EXE 程序，连接程序会自动生成特定的格式，当将其存储到磁盘上时，将 512 字节

或更长的专门的标题块加到它前面(第 23 章提供标题块的细节)。

也可以编写可执行的.COM 程序。.COM 程序的优点是它们比差不多的.EXE 程序小,并且更适于作为常驻程序。.COM 格式在早期的微计算机中是有其根源的,那时程序规模仅限于 64K,因而比较原始并且受到限制。

5.9.1 .EXE 程序与.COM 程序的区别

作为.EXE 执行的程序与作为.COM 执行的程序之间的重要区别包括程序的规模,分段,以及初始化。

1. 程序规模

.COM 程序的指令和数据都使用同一个段,基本上被限制为最大值 64K,其中包括程序段前缀(PSP)。PSP 是一个 256 字节(100H)块,由装入程序直接插在.COM 和.EXE 程序的前面,这是在装入程序把它们从磁盘装入到存储器的时候进行的。

.COM 程序总是比与它相对应的.EXE 程序要小,原因之一是磁盘上放在.EXE 程序前面的 512 字节的标题记录是不放在.COM 程序前面的(不能混淆标题记录,在第 23 章 PSP 部分会涉及这个问题)。.COM 程序是可执行程序的绝对映像,它不具有浮动地址信息。

2. 段

.COM 程序段的使用是它和.EXE 程序的重要的(而且是比较容易的)区别。完整的.COM 程序把 PSP,堆栈,数据段和代码段组合在一个代码段中。

对于.EXE 程序,通常要定义数据段并且用该段的地址初始化 DS。对于.COM 程序,在代码段内定义数据,就像在第 3 章里使用 DEBUG 时所做的那样。

尽管必须为.EXE 程序定义堆栈段,但汇编程序会自动地为.COM 程序生成堆栈。因此,编写要被转换成.COM 格式的程序时,定义堆栈是可以省略的。

如果对一个程序而言,64K 的段是足够大的,那么装入程序会在段的末尾设置堆栈并用栈顶地址设置 SP 寄存器。如果 64K 的段不是足够大,则汇编程序会在程序之后较高端的存储器中建立堆栈。(然而,这么大的程序应该编写成.EXE 格式。)

在本书中,许多较小的程序都是.COM 格式,它们很容易和.EXE 格式区别开来。图 5-6 比较了.EXE 和.COM 格式的段寻址。

3. 初始化

当装入程序把要执行的.COM 程序装入时,它自动地用 PSP 的地址初始化 CS、DS、SS 和 ES。由于 CS 和 DS 现在的内容是指令执行时的正确的初始段地址,所以.COM 程序不必把它们初始化。

因为 PSP 的大小是 100H 个字节,寻址是从 100H 字节的偏移值处开始的,所以必须编写伪操作 ORG 100H,紧跟在代码段的 SEGMENT 或.CODE 语句之后。ORG 伪操作告诉汇编程序把位置计数器设置在 100H。然后,汇编程序在距 PSP 的起点 100H 字节的偏移值处产生目标码,在那里开始.COM 程序的编码。

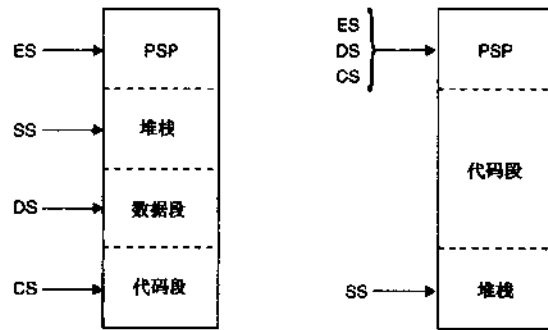


图 5-6 .EXE 和 .COM 的段

5.9.2 编写.COM 格式的程序

如果源程序是按 .EXE 格式编写的，那么可以使用编辑程序把指令转换成 .COM 格式。

图 5-7 的程序是类似于图 5-2 的程序，但现在修改成符合 .COM 的要求了。注意以下特点：

- 没有定义堆栈或数据段。
- ASSUME 语句告诉汇编程序当程序为执行而装入时，CS、SS、DS 和 ES 将包含代码段的起始地址(这里是 PSP 的起点)。
- 伪操作 ORG 100H 告诉汇编程序使位置计数器从 PSP 起点前进 100H 字节。当装入程序装入 .COM 程序时，把 100H 存放到 IP(指令指针)寄存器。由于这一特点，跟在 ORG 后面的第一个语句必须是可执行的指令。
- JMP 指令把执行的控制权绕过所定义的数据往下传送。有些程序员把数据项放在指令的后面，这样就不需要开始的 JMP 指令了。一开始就编写数据项，可以略微加速汇编的处理，除此之外没有其他优点。在本书所举的例子中，先定义数据仅仅是一种程序设计习惯而已。
- BEGIN 和 MAIN 仅仅是描述性的标号，对于汇编程序没有其他意义。对于这些标号，你可以使用任何有效的名字。
- 标准的 INT 21H 的 4CH 功能结束处理过程。

```

TITLE      A05COM1 COM program to move and add
CODESEG    SEGMENT PARA 'Code'
            ASSUME  CS:CODESEG,DS:CODESEG,SS:CODESEG,ES:CODESEG
            ORG     100H           ; 在 PSP 末端开始
BEGIN:     JMP     MAIN           ; 绕过数据转移
; -----
FLDD       DW      215            ; 定义数据
FLDE       DW      125
FLDF       DW      ?
; -----
MAIN       PROC     NEAR
            MOV     AX,FLDD        ; 把 0215 传送到 AX
            ADD     AX,FLDE        ; 把 0125 加到 AX 中
            MOV     FLDF,AX       ; 在 FLDF 中存放和
            MOV     AX,4C00H      ; 结束处理
            INT     21H
MAIN       ENDP
CODESEG    ENDS
            END      BEGIN
  
```

图 5-7 具有常规段的 .COM 源程序

5.9.3 把.EXE 格式转换成.COM 格式

由于汇编程序版本不同，产生.COM 文件的方法也不一样：

- Microsoft MASM 6.1 的 ML 命令汇编、连接并转换成 .COM 程序，如果该程序指定的是 Tiny 存储模型，用这一个命令就可以了。
- Microsoft MASM 5.1 产生.OBJ 文件，然后把它加以连接而生成 .EXE 文件。为了把 .EXE 文件转换成 .COM 文件，使用名为 EXE2BIN 的程序。程序名的意思是“把 EXE 转换为 BIN”，其中 BIN 指的是二进制文件，但指定输出文件扩展名是.COM。
- TASM 允许用 TLINK 程序来建立 .COM 文件。

有关转换成.COM 格式的细节可参阅附录 E。

当完成.COM 格式的转换后，可以删除已产生的.OBJ 和.EXE 文件。.EXE 和.COM 程序的大小分别是 792 个字节和 24 个字节。差别是很大的，这是由存放在 .EXE 模块开始处的 512 字节的标题块造成的。

图 5-8 表示的是使用简化段伪操作去编写.COM 程序。重复一次，只定义了代码段而没有定义堆栈或数据段。对于 MASM 6.1，这种程序应该使用 Tiny 存储模型。

调试忠告。忽略了 .COM 的任何一个要求，都可能导致程序的失败。例如，如果 EXE2BIN 发现错误，它只是简单地通知你该文件不能转换，而不提供任何理由。检查 SEGMENT，ASSUME，以及 END 语句。如果遗漏了 ORG 100H，正在执行的程序会错误地访问 PSP 中的数据，造成无法预料的后果。

如果在 DEBUG 下运行 .COM 程序，用 D CS:100 查看数据和指令。

试图执行 .COM 程序中的 .EXE 模块，是会失败的，一定要删除这个文件。

TITLE	A05COM2	COM program to move and add data
	.MODEL	TINY
	.CODE	
	ORG	100H ; 在 PSP 末端开始
BEGIN:	JMP	MAIN ; 绕过数据转移

FLDD	DW	215 ; 定义数据
FLDE	DW	125
FLDF	DW	?

MAIN	PROC	NEAR
	MOV	AX, FLDD ; 把 0215 传送到 AX
	ADD	AX, FLDE ; 把 0125 加到 AX 中
	MOV	FLDF, AX ; 在 FLDF 中存放和
	MOV	AX, 4C00H ; 结束处理
	INT	21H
MAIN	ENDP	
	END	BEGIN

图 5-8 使用简化段伪操作的.COM 程序

5.10 要 点

- 汇编程序把源程序转换成.OBJ 文件并生成可选的列表文件和交叉引用文件。

- 跟在汇编程序列表之后的段与组表，显示在程序中所定义的任何段与组。符号表显示出所有的符号(数据名和指令标号)。
- 连接程序把.OBJ文件转换成可执行的.EXE文件。
- 简化段伪操作为数据段产生名字 `_DATA`，为堆栈段产生 `STACK`，为代码段产生 `_TEXT`，并产生许多预定义的等式。
- 交叉引用表对于定位所有对数据项引用是有用的。
- .COM程序被限制为一个64K的段，并比与它相对应的.EXE程序要小。这种程序不需要定义堆栈或数据段，也不需要初始化DS寄存器。
- 编写成.COM的程序要求在代码段的 `SEGMENT` 语句之后，紧跟着 `ORG 100H`。该语句把开始执行的偏移地址设置在PSP之后。
- 系统把.COM程序的堆栈建立在程序的末尾。

5.11 习 题

5-1. 编写命令行去汇编命名为 `MONITOR.ASM` 的源程序，要有列表、目标和交叉引用各文件。

5-2. 编写命令行去连接 `MONITOR.OBJ`(习题5-1中产生的)，采用单独的连接命令。

5-3. 为5-2题中生成的 `MONITOR.EXE` 编写如下要求的命令：(a)直接从DOS执行，(b)通过DEBUG执行。

5-4. 说明以下每一种文件的用途：(a).ASM文件，(b).LST文件，(c).MAP文件，(d).CRF文件，(e).OBJ文件，(f).EXE文件。

5-5. 采用常规段定义并且用 `DATASEG` 作为数据段的名称，编写2条MOV指令初始化DS寄存器。

5-6. 用常规段定义编写一个汇编程序，要求如下：(a)把立即值 `hex 40` 传送到AL寄存器，(b)将AL的内容左移一位(`SHL AL, 1`)，(c)把立即值 `hex 1A` 传送到BL，(d)AL乘以BL(`MUL BL`)。不要遗忘结束程序执行的指令。程序不需要定义或初始化数据段。复制框架程序并用编辑程序去开发该程序。汇编，连接并使用DEBUG跟踪和检查代码段与寄存器。

5-7. 把5-6题修改成用简化段伪操作。汇编和连接该程序，并和原始程序比较目标码、符号表和连接映像。使用DEBUG跟踪并检查代码段和寄存器。

5-8. 按以下要求把数据段加到5-6题的程序中：

- 定义名为 `ITEMA` 的1字节项(DB)，内容为 `hex 40`；定义另一个名为 `ITEMB` 的1字节项(DB)，内容为 `hex 1A`。
- 定义2字节的项(DW)，名为 `ITEMC`，不带常数。
- 把 `ITEMA` 的内容传送到AL并左移一位。
- AL乘以 `ITEMB`(`MUL ITEMB`)。
- 把AX中的乘积传送到 `ITEMC`。

汇编，连接，并使用DEBUG检查该程序。

5-9. 用简化段伪操作修改5-8题的程序。汇编、连接该程序，并和原始程序比较目标码、

符号表和连接映像。使用 DEBUG 检查该程序。

5-10. 对于以下每个数据项, 指出汇编程序的位置计数器的内容:

```
0000 WORD1 DW 0
----- WORD2 DW 0
----- BYTE1 DB 0
                EVEN
----- WORD3 DW 0
----- BYTE2 DB 0
```

5-11. .COM 程序的最大规模是多少?

5-12. 为了把源程序转换为.COM 格式, 你可以定义哪些段?

5-13. 解释为什么要在被转换的.COM 格式程序的起点编写 ORG 100H。

5-14. 为什么为.COM 程序定义堆栈是没有必要的?

5-15. 把 5-8 题(常规段)中的程序修改成.COM 格式。汇编、连接, 把它转换成.COM, 并在 DEBUG 下执行它。

5-16. 把 5-9 题(简化段)中的程序修改成.COM 格式。汇编、连接, 把它转换成.COM, 并在 DEBUG 下执行它(对于 MASM 6.x, 使用 Tiny 存储模型)。

5-17. 下面的程序中含有许多汇编期间的错误, 如右边所示。请改正每个错误。

```
TITLE      A04ASM2 (EXE) program errors

.MODEL     SMALL
.STACK     64
.DATA

DATA1      DB      25
DATA2      DB      280          ; 1: 值超出范围
DATA3      DW      ?

CODE

MAIN       PROC

    MOV     AX, data      ; 2: 操作数类型不适当
    MOV     DS, AX
    MOV     AX, DATA1    ; 3: 操作数类型必须匹配
    ADD     AX, DATA2    ; 4: 操作数类型必须匹配
    MOV     DATA3, AX
    MOV     FX, 4C00H     ; 5: 符号没有定义
    INT     21H

MAIN       ENDP

END        MAIN
```

汇编程序没有定位的有关 MAIN PROC 语句的另一个错误, 是什么?

目的：提供汇编语言指令系统和对于寻址数据要求的基础

6.1 引言

本章介绍处理器指令系统的类别。正式包括在本章的指令是 MOV、MOVSX、MOVZX、XCHG、LEA、INC、DEC、ADD、SUB 和 INT，还有作为立即值使用的指令操作数中的常数。最后，本章讨论贯穿本书其余部分所使用的基本寻址格式，然后说明地址对准和段跨越前缀。

6.2 符号指令系统——概述

以下是 Intel 处理器系列所用的符号指令表，它是按类别排列的。虽然这个表似乎很难对付，但其中许多指令是很少用到的。

1. 算术运算

ADC:	带进位加法	INC:	加 1
ADD:	二进制数加法	MUL:	无符号数乘法
DEC:	减 1	NEG:	求补
DIV:	无符号数除法	SBB:	带借位减法
IDIV:	带符号数(整数)除法	SUB:	二进制减法
IMUL:	带符号数(整数)乘法	XADD:	交换并相加

2. ASCII-BCD 转换

AAA:	加后 ASCII 调整	AAS:	减后 ASCII 调整
AAD:	除前 ASCII 调整	DAA:	加后十进制调整
AAM:	乘后 ASCII 调整	DAS:	减后十进制调整

3. 移位

RCL:	带进位循环左移	SAR:	算术右移
------	---------	------	------

RCR: 带进位循环右移
 ROL: 循环左移
 ROR: 循环右移
 SAL: 算术左移

SHL: 逻辑左移
 SHR: 逻辑右移
 SHLD: 双精度左移(80386+)
 SHRD: 双精度右移(80386+)

4. 比较

BSF/BSR: 位扫描(80386+)
 BT/BTC/BTR/BTS: 位测试(80386+)
 CMP: 比较
 CMPSn: 串比较

CMPXCHG: 比较并交换(80486+)
 CMPXCHG8B: 比较并交换(pentium+)
 TEST: 测试位

5. 数据传送

LDS: 装入数据段寄存器
 LEA: 装入有效地址
 LES: 装入附加段寄存器
 LODS: 从串取
 LSS: 装入堆栈段寄存器
 MOV: 传送数据

MOVS: 串传送
 MOVSB: 带符号扩展传送
 MOVZX: 带零扩展传送
 STOS: 存入串
 XCHG: 交换
 XLAT: 换码

6. 标志操作

CLC: 清除进位标志
 CLD: 清除方向标志
 CLI: 清除中断标志
 CMC: 进位标志求反
 LAHF: 标志送 AH
 POPF: 标志出栈

PUSHF: 标志进栈
 SAHF: AH 送标志寄存器
 STC: 进位标志置 1
 CTD: 方向标志置 1
 STI: 中断标志置 1

7. 输入/输出

IN: 输入字节或字
 INSn: 串输入(80286+)

OUT: 输出字节或字
 OUTSn: 串输出(80286+)

8. 逻辑操作

AND: 逻辑与
 NOT: 逻辑非

OR: 逻辑或
 XOR: 异或

9. 循环

LOOP: 循环直到完成
 LOOPE: 相等时循环

LOOPNZ: 不为零时循环
 LOOPNEW: 不相等时循环(80386+)

LOOPZ: 为零时循环
LOOPNE: 不相等时循环

LOOPNZ: 不为零时循环(80386+)

10. 处理器控制

HLT: 进入暂停状态
LOCK: 封锁总线

NOP: 无操作
WAIT: 置处理器于等待状态

11. 堆栈操作

ENTER: 建立堆栈帧(80286+)
LEAVE: 结束堆栈帧(80286+)
POP: 字出栈
POPF: 标志出栈

POPA: 所有通用寄存器出栈(80286+)
PUSH: 字进栈
PUSHA: 所有通用寄存器进栈(80286+)
PUSHF: 标志进栈

12. 串操作

CMPS: 串比较
LODS: 从串取
MOVS: 串传送
REP: 串重复
REPE: 相等时重复

REPZ: 为零时重复
REPNE: 不相等时重复
REPNZ: 不为零时重复
SCAS: 串扫描
STOS: 存入串

13. 转移(条件)

INTO: 溢出中断
JA: 若高于则转移
JAE: 若高于或等于则转移
JB: 若低于则转移
JBE: 若低于或等于则转移
JC: 若进位为1则转移
JCXZ: 若CX为零则转移
JE: 若相等则转移
JG: 若大于则转移
JGE: 若大于或等于则转移
JL: 若小于则转移
JLE: 若小于或等于则转移
JNA: 若不高于则转移
JNAE: 若不高于或等于则转移
JNB: 若不低于则转移
JNBE: 若不低于或等于则转移

JNC: 若进位为零则转移
JNE: 若不相等则转移
JNG: 若不大于则转移
JNGE: 若不大于或等于则转移
JNL: 若不小于则转移
JNLE: 若不小于或等于则转移
JNO: 若不溢出则转移
JNP: 若奇偶位为0则转移
JNS: 若结果为正则转移
JNZ: 若结果不为零则转移
JO: 若溢出则转移
JP: 若奇偶位为1则转移
JPE: 若奇偶性为偶则转移
JPO: 若奇偶性为奇则转移
JS: 若结果为负则转移
JZ: 若结果为零则转移

14. 转移(无条件)

CALL:	调用过程	JMP:	无条件转移
INT:	中断	RET:	返回
IRET:	中断返回	RETN/RETF:	近返回/远返回

15. 类型转换

CBW:	字节转换为字
CDQ:	双字转换为四字(80386+)
CWD:	字转换为双字
CWDE:	字转换为扩展的双字(80386+)

标明处理器的指令(如 80386+)要求使用处理器伪操作(第 3 章涉及此内容), 以便正确地进行汇编。

6.3 数据传送指令

这一节讨论与数据传送有关的一些通用指令。

6.3.1 MOV 指令

MOV 指令是把第二个操作数的地址所引用的数据传送(或复制)到第一个操作数的地址中。发送字段是不变的。引用存储器或寄存器的操作数大小必须一致(必须两个都是字节、字或双字)。MOV 的格式是:

[label:]	MOV	register/memory, register/memory/immediate
-----------	-----	--

(Label: 标号, register: 寄存器, memory: 存储器, immediate: 立即数)

下面是按类型分的 4 个合法 MOV 操作的例子, 给定以下数据项:

```
BYTEFLD DB ? ; 定义字节
WORDFLD DW ? ; 定义字
```

1. 寄存器传送

MOV EDX, ECX	: 寄存器到寄存器
MOV ES, AX	: 寄存器到段寄存器
MOV BYTEFLD, DH	: 寄存器到存储器, 直接。
MOV [DI], BX	: 寄存器到存储器, 间接。

2. 立即数传送

MOV CX, 40H	: 立即数到寄存器
MOV BYTEFLD, 25	: 立即数到存储器, 直接。
MOV WORDFLD[BX], 16H	: 立即数到存储器, 间接。

3. 直接存储器传送

```
MOV CH, BYTEFLD          ; 存储器到寄存器, 直接。
MOV CX, WORDFLD[BX]      ; 存储器到寄存器, 间接。
```

4. 段寄存器传送

```
MOV AX, DS                ; 段寄存器到寄存器
MOV WORDFLD, DS           ; 段寄存器到存储器
```

可以向寄存器传送字节(MOV CH, BYTEFLD), 字(MOV CX, WORDFLD), 或双字(MOV ECX, DWORDFLD)。操作数只影响所引用的那一部分寄存器, 例如, 向 CL 传送字节对 CH 没有影响。无效的 MOV 操作包括以下一些:

```
MOV DL, WORD_VAL          ; 字到字节
MOV CX, BYTE_VAL          ; 字节到字
MOV WORD_VAL, EBX         ; 双字到字
MOV BYTE_VAL2, BYTE_VAL1  ; 存储器到存储器
MOV ES, 225               ; 立即数到段寄存器
MOV ES, DS                ; 段寄存器到段寄存器
```

完成这些操作需要一条以上的指令。

6.3.2 传送并填充指令: MOVZX 与 MOVZX

对于 MOV 指令, 目的必须和源的长度相同, 比如字节到字节, 字到字。MOVZX 和 MOVZX(传送并填充)指令(80386+)使传送的数据可以从字节或字的源到字或双字的目的地。下面是它们的格式:

[label:]	MOVZX/MOVZX	register/memory, register/memory/immediate
-----------	-------------	--

MOVZX 用于带符号的算术值, 把字节或字传送到字或双字的目的地, 并把符号位(源的最左边的一位)填入该目的地的最左边的所有位。MOVZX 用于无符号的数字值, 把字节或字传送到字或双字的目的地, 并在该目的地的最左边所有位填入零。下面这个例子, 考虑把内容为 10110000 的字节传送到字, 目的字的结果取决于所选择的指令:

```
MOVZX CX, 10110000B        ; CX=11111111 10110000
MOVZX CX, 10110000B        ; CX=00000000 10110000
```

下面是使用 MOVZX 和 MOVZX 的一些其他例子:

```
BYTE1 DB 25                ; 字节
WORD1 DW 40                ; 字
DWORD1 DD 160              ; 双字
.386 ...

MOVZX CX, BYTE1            ; 字节到字
MOVZX WORD1, BH            ; 字节到字
MOVZX EBX, WORD1          ; 字到双字
MOVZX DWORD1, CX          ; 字到双字
```

.386 处理器伪操作告诉汇编程序接受由 80386 所引入的指令。第 7 章和第 12 章会涉及有

关带符号与无符号数据的细节。

6.3.3 XCHG 指令

XCHG 完成另一种类型的数据传送，不是简单地把数据从一个单元复制到另一个单元，XCHG 要交换两个数据项。XCHG 的格式是：

[Label:]	XCHG	Register/memory, register/memory
-----------	------	----------------------------------

有效的 XCHG 操作是要在两个寄存器之间或是寄存器与存储器之间交换数据。以下是两个例子：

```
WORDQ DW ?           ; 字数据项
...
XCHG CL, BH          ; 交换两个寄存器的内容
XCHG CX, WORDQ       ; 交换寄存器和存储器的内容
```

6.3.4 LEA 指令

LEA 对于用偏移地址去初始化寄存器是有用的。其格式是：

[Label:]	LEA	register, memory
-----------	-----	------------------

LEA 通常的用法是用偏移值来初始化 BX、DI 或 SI，为的是指向存储器中的一个地址，本书从头到尾都是这么做的。下面是一个例子：

```
DATATBL DB 25 DUP(?)   ; 25 个字节的表
BYTEFLD DB ?          ; 一个字节
...
LEA BX, DATATBL        ; 装入偏移地址
MOV BYTEFLD, [BX]      ; 传送 DATATBL 的第一个字节
```

与 LEA 等效的操作是带 OFFSET 操作符的 MOV，它产生的机器码稍短一些，像下面这样使用：

```
MOV BX, OFFSET DATATBL ; 装入偏移地址
```

6.4 基本算术指令

这一节讨论有关加法与减法的基本指令：INC，DEC，ADD，以及 SUB。

6.4.1 INC 与 DEC 指令

INC 与 DEC 是使寄存器和存储器单元的内容加 1 或减 1 的指令。INC 与 DEC 的格式是：

[label:]	INC/DEC	register/memory
-----------	---------	-----------------

注意: INC 与 DEC 只需要一个操作数。根据运算的结果来确定是清除或设置 OF(有到符号位的进位, 无进位输出), SF(正/负), 以及 ZF(零/非零)标志。条件转移指令可以测试这些条件。

如果字节值是 FFH, INC 把它“加 1”到 00H 并置 SF 为 0(结果为正)及 ZF 为 1(结果为零)。DEC 把 00H “减 1”为 FFH 并置 SF 为 1(结果为负)及 ZF 为 0(结果为非零)。

6.4.2 ADD 与 SUB 指令

这一节概括地讨论 ADD 与 SUB, 第 12 章会更详细地加以说明。它们的格式是:

[label,]	ADD/SUB	register/memory, register/memory/immediate
-----------	---------	--

有效的操作包括从寄存器到寄存器, 从寄存器到存储器, 从存储器到寄存器, 从立即数到寄存器, 以及从立即数到存储器。下面是一些例子:

ADD AX, CX	; 寄存器与寄存器相加
ADD EBX, DBLWORD	; 存储器双字加到寄存器
SUB BL, 10	; 从寄存器中减去立即数

受影响的标志是 AF, CF, OF, PF, SF, 以及 ZF。例如, 结果为零时 ZF 置 1, 而结果为负时 SF 置 1。

6.5 重复传送操作

到此为止, 程序已经涉及了把立即数据传送到寄存器, 把所定义的存储器中的数据传送到寄存器, 把寄存器内容传送到存储器, 以及把一个寄存器的内容传送到另一个寄存器。在所有情况下, 数据长度被限定为 1 个、2 个或 4 个字节, 并且没有把来自一个存储区的数据直接传送到另一个存储区的操作。这一节说明如何一个字节跟着一个字节地传送整个数据项。另一种使用串指令的方法会在第 11 章叙述。

图 6-1 的程序中, 数据段包含定义为 HEADNG1 和 HEADNG2 的两个 9 字节字段。程序的目标是把 HEADNG1 的内容从左到右传送到 HEADNG2。由于这些字段每个都是 9 个字节长, 比简单的 MOV 指令所要求的要多。该程序有不少新特点。

为了一步步地传送 HEADNG1 和 HEADNG2, 该程序把 CX 初始化为 9(这两个字段的长度)并用 SI 和 DI 作变址。两条 LEA 指令把 HEADNG1 和 HEADNG2 的偏移地址装入到 SI 和 DI 中, 如下所示:

LEA SI, HEADNG1	; 初始化 HEADNG1 和 HEADNG2 的偏移地址
LEA DI, HEADNG2	

程序用 SI 和 DI 中的地址, 把 HEADNG1 的第一个字节传送到 HEADNG2 的第一个字节。MOV 指令操作数中的带方括号的 SI 和 DI 指的是间接寻址: 指令是用给定寄存器中的偏移地址来访问存储单元的。因此, MOV AL,[SI]的意思是“使用 SI 中的偏移地址(HEADNG1+0)把所引用的字节传送到 AL”。指令 MOV [DI],AL 的意思是“把 AL 的内容传送到由 DI 所

引用的偏移地址(HEADNG2+0)”。该程序要把这两条指令重复 9 次，一次有一个字符传送到各自的字段中。为此目的，程序使用了还没有说明过的条件转移指令 JNZ(若非零则转移)。

```

TITLE      A06MOVE (EXE)  Repetitive move operations
.MODEL     SMALL
.STACK     64

;-----
.DATA
HEADNG1 DB 'InterTech'
HEADNG2 DB 9 DUP('*'), '$'
;-----
.CODE
A10MAIN PROC FAR
MOV AX,@data ; 初始化
MOV DS,AX ; 段寄存器
MOV ES,AX

MOV CX,09 ; 初始化传送9个字符
LEA SI,HEADNG1 ; 初始化 HEADNG1 的
LEA DI,HEADNG2 ; 偏移地址

A20:
MOV AL,[SI] ; 从 HEADNG1 取字符,
MOV [DI],AL ; 传送到 HEADNG2
INC SI ; 加1指向下一个 HEADNG1 中的字符
INC DI ; 加1指向下一个 HEADNG2 中的位置
DEC CX ; 循环计数减1
JNZ A20 ; 计数不为零? 是, 循环
; 结束
MOV AH,09H ; 请求显示
LEA DX,HEADNG2 ; HEADNG2
INT 21H

MOV AX,4C00H ; 结束处理
INT 21H

A10MAIN ENDP
END A10MAIN
    
```

图 6-1 重复传送操作

两条 INC 指令使 SI 和 DI 加 1，DEC 指令使 CX 减 1。DEC 还要设置或清除零标志，这取决于 CX 的结果；如果该结果是非零，则仍有更多的字符要传送，并且 JNZ 转移返回到标号 A20 去重复传送指令。由于 SI 与 DI 已被加 1，所以下一条 MOV 指令引用 HEADNG1+1 和 HEADNG2+1。循环以这种方式继续下去直到它把总共 9 个字符传送完，到达 HEADNG2+8 为止。

同样，程序需要用指令在处理结束后显示 HEADNG2 的内容：①在 AH 中装入 09H 功能要求显示；②在 DX 中装入 HEADNG2 的地址；③执行指令 INT 21H。

INT 操作显示从 HEADNG2 第一个字节开始到结束的 '\$' 符号为止的全部字符。'\$' 是紧跟在 HEADNG2 定义之后的，这一操作更详细的说明见第 8 章。

作为练习，键入这一程序，对它进行汇编和连接，并且使用 DEBUG 跟踪它的运行。注意对于堆栈，各寄存器，以及 IP(特别是在 JNZ 执行之后)的影响。利用 D DS:0 查看 HEADNG2 的变化。

6.6 INT 指令

INT 指令能中断程序本身的处理过程，例如初始化鼠标驱动程序：

```

MOV AX,00H ; 初始化
    
```


INT 33H

; 鼠标驱动程序

INT 退出正常处理过程并访问位于低端存储器的中断向量表, 以便确定所需要请求例程的地址。然后该操作传送给 BIOS 或操作系统去完成指定的动作并返回程序继续原来的处理过程。通常, 中断必须完成输入或输出操作的许多复杂步骤。中断需要能方便地退出程序, 并且在成功完成指定动作后, 又立即返回该程序。为了这一目的, INT 执行以下操作:

- 标志寄存器的内容进栈(堆栈指针减 2)。
- 清除中断与陷阱标志。
- CS 寄存器进栈
- 指令指针(内有下一条指令的地址)进栈。
- 完成所要求的操作。

为了从中断返回, 发送一条 IRET(中断返回), 使寄存器出栈。所恢复的 CS:IP 使之返回到紧跟在 INT 后的指令。

由于上述处理是完全自动化的, 所以唯一要做的是按照进栈、出栈以及使用适当的 INT 操作的需要, 定义一个足够大的堆栈。

6.7 寻址方式

操作数地址为指令的处理提供数据来源。某些指令, 如 CLC 和 RET, 不需要操作数; 而其它指令可能有 1 个、2 个或 3 个操作数。其中有 2 个操作数的, 第一个操作数是目的, 它包含的数据在寄存器或存储器中, 并且是要被处理的。第二个操作数是源, 它包含或是要被提交的数据(立即数)或是数据的地址(在存储器中的或寄存器的)。对于大多数指令, 源数据是不被操作改变的。寻址的 3 个基本方式是: 寄存器寻址方式, 立即寻址方式, 以及存储器寻址方式, 存储器寻址由 6 种类型组成, 总共有 8 种方式。

1. 寄存器寻址

对于这种方式, 寄存器提供 8 位、16 位或 32 位寄存器中任何一个的名字。由指令决定, 寄存器可以出现在第一个操作数中, 第二个操作数中, 或者两个操作数中都出现。如以下例子所示:

```
MOV  DX, WORD_MEM      ; 寄存器在第一个操作数中
MOV  WORD_MEM, CX       ; 寄存器在第二个操作数中
MOV  EDX, EBX           ; 寄存器在两个操作数中
```

由于在寄存器之间处理数据不涉及引用存储器, 所以是最快的操作类型。

2. 立即寻址

立即操作数包括常数值或表达式。下面是一些有效的立即常数的例子:

```
十六进制:      0148H
十进制:        328(汇编程序把它转换成 0148H)
二进制:       101001000B(可转换成 0148H)
```

对于许多有 2 个操作数的指令，第一个操作数可以是个寄存器或存储单元，而第二个操作数可以是立即常数。目的字段(第一个操作数)定义数据的长度。下面是一些例子：

```

BYTE_VAL DB 150           ; 定义字节
WORD_VAL DW 300           ; 字
DBWD_VAL DD 0             ; 双字
...
SUB BYTE_VAL, 50           ; 立即数到存储器(字节)
MOV WORD_VAL, 40H          ; 立即数到存储器(字)
MOV DBWD_VAL, 0            ; 立即数到存储器(双字)
MOV AX, 0245H              ; 立即数到寄存器(字)

```

最后一个例子中的指令是把立即常数 0245H 传送到 AX。3 字节的目标码是 B84502，其中 B8 的意思是“把立即值传送到 AX”，而紧跟的 2 个字节是值本身(4502H，按字节相反顺序排列)。

立即操作数的使用提供了更高的处理速度，它在数据段中定义数字常数并在操作数中引用它的处理速度要快。

立即常数的长度不能超过由第一个操作数所定义的长度。下面的无效的例子中，立即操作数是两个字节，而 AL 只是一个字节：

```
MOV AL, 0245H              ; 无效的立即数长度
```

但是，如果立即操作数比要接收的操作数短，如在下面的指令中

```
ADD AX, 48H                ; 有效的立即数长度
```

则汇编程序把立即操作数扩展成两个字节的 0048H，并把它存放在目标码中为 4800H。

3. 直接存储器寻址

在这一格式中，一个操作数引用存储单元，而另一个操作数则是引用寄存器(允许 2 个操作数直接访问存储器的指令只有 MOV 和 CMPS)。DS 是访问存储器中数据的默认段寄存器，如 DS：偏移值。以下是一些例子：

```

ADD BYTE_VAL, DL           ; 寄存器加到存储器(字节)
MOV BX, WORD_VAL           ; 存储器传送到寄存器(字)

```

4. 直接-偏移值寻址

这种寻址方式是直接寻址方式的变种，它使用算术操作符修改地址。以下例子是使用这些定义的表格：

```

BYTE_TBL DB 12, 15, 16, 22, ... ; 字节表
WORD_TBL DW 163, 227, 485, ...   ; 字表
DBWD_TBL DD 465, 563, 897, ...   ; 双字表

```

(1) 字节操作。这些指令访问 BYTE_TBL 中的字节：

```

MOV CL, BYTE_TBL[2]         ; 从 BYTE_TBL 中取得字节
MOV CL, BYTE_TBL+2          ; 同样的操作

```

第一条 MOV 使用算术操作符访问 BYTE_TBL 的第三个字节(16)(BYTE_TBL[0]是第一个字节，BYTE_TBL[1]是第二个字节，而 BYTE_TBL[2]则是第三个字节)。第二条 MOV 使用加

(+)操作符, 效果完全一样。

(2) 字操作。这些指令访问 WORD_TBL 中的字:

```
MOV CX, WORD_TBL[4]           ; 从 WORD_TBL 中取得字
MOV CX, WORD_TBL+4            ; 同样的操作
```

MOV 访问 WORD_TBL 的第三个字(WORD_TBL[0]是第一个字, WORD_TBL[2]是第二个字, 而 WORD_TBL[4]是第三个字)。

(3) 双字操作。这些指令访问 DBWD_TBL 中的双字:

```
MOV CX, DBWD_TBL[8]           ; 从 DBWD_TBL 取得双字
MOV CX, DBWD_TBL+8            ; 同样的操作
```

MOV 访问 DBWD_TBL 的第三个双字(DBWD_TBL[0]是第一个双字, DBWD_TBL[4]是第二个双字, 而 DBWD_TBL[8]是第三个双字)。

5. 间接存储器寻址

间接寻址利用了计算机的段: 偏移值的寻址能力。用于此目的的寄存器是基址寄存器(BX 或 BP)和变址寄存器(DI 和 SI), 方括号内的编码是指明对存储器的引用。如果使用了.386, .486 或.586 伪操作, 还可以使用任何通用寄存器(EAX, EBX, ECX, 以及 EDX)去做间接寻址。

如[DI]那样的间接寻址告诉汇编程序, 当程序接着执行时, 所用的存储器地址将在 DI 中。为了处理数据段中的数据, BX、DI 和 SI 与 DS 联系在一起成为 DS:BX, DS:DI 和 DS:SI。BP 和 SS 联系在一起成为 SS:BP, 用来处理堆栈中的数据。

当第一个操作数包含间接地址时, 第二个操作数引用寄存器或立即值; 当第二个操作数包含间接地址时, 第一个操作数引用寄存器。注意方括号中的 BP, BX, DI, 或 SI 的引用是指间接操作数, 当程序执行时, 处理器是把这些寄存器的内容作为偏移地址处理的。

下面的例子中, 首先 LEA 用 DATA_VAL 的偏移地址初始化 BX。然后 MOV 把 CL 存入由现在 BX 中的地址所指向的存储单元, 在这种情况下, 是 DATA_VAL:

```
DATA_VAL, DB 50                ; 定义字节
...
LEA BX, DATA_VAL              ; 用偏移地址装入 BX
MOV [BX], CL                   ; 把 CL 传送到 DATA_VAL
```

这两条指令和 MOV DATA_VAL, CL 的效果是一样的。但是, 变址寻址方式是比较重要的。这里再举几个例子:

```
ADD CL, [BX]                   ; 第 2 个操作数=DS:BX
MOV BYTE PTR [DI], 25          ; 第 1 个操作数=DS:DI
ADD [BP], CL                   ; 第 1 个操作数=SS:BP
.386
MOV DX, [EAX]                  ; 第 2 个操作数=DS:EAX
```

下面这个例子使用偏移的绝对值:

```
MOV CX, DS:[38B0H]             ; 存储器偏移值 38B0H 中的字
```

6. 基址位移量寻址

这一寻址方式还是使用基址寄存器(BX 和 BP)和变址寄存器(DI 和 SI), 但与一个位移量(数或偏移值)组合起来形成有效地址。下面的 MOV 指令把零传送到紧跟在 DATA_TBL 起点后的 2 个字节的单元中:

```
DATA_TBL DB 365 DUP(?)      ; 定义字节
...
LEA BX, DATA_TBL          ; 把偏移值装入 BX
MOV BYTE PTR [BX+2], 0     ; 把 0 传送到 DATA_TBL+2
```

一些其他的例子如下:

```
ADD CL, [DI+12]             ; DI 内的偏移值加 12 (或 12[DI])
SUB DATA_TBL[SI], 25       ; SI 的内容为偏移值 (0-364)
MOV DATA_TBL[DI], DL       ; DI 的内容为偏移值 (0-364)
.386
MOV DX, [EAX+4]             ; EAX 内的偏移值加 4
ADD DATA_TBL[EDX], CL      ; EDX+偏移值 DATA_TBL
```

7. 基址-变址寻址

这种寻址方式是把基址寄存器(BX 或 BP)和变址寄存器(DI 或 SI)组合起来形成有效地址。例如, [BX+DI]指的是 BX 中的地址加上 DI 中的地址。这种方式通常用于对二维数组的寻址, 比如 BX 引用的是行, 而 SI 则是列。下面是一些例子:

```
MOV AX, [BX+SI]             ; 从存储器传送字
ADD [BX-DI], CL             ; 把字节加到存储器
```

8. 带位移量的基址-变址寻址

这一寻址方式是基址-变址寻址方式的变种, 它由基址寄存器, 变址寄存器和位移量组合, 形成有效地址。一些例子如下:

```
MOV AX, [BX+DI+10]          ; 或 10[BX+DI]
MOV CL, DATA_TBL[BX+DI]    ; 或 [BX+DI+DATA_TBL]
```

6.8 段跨越前缀

处理器在寻址时会自动地选择适当的段: 为取指令用 CS:IP, 为存取存储器的数据用 DS:偏移值, 为访问堆栈用 SS:SP。有这样的時候, 特别是对于大型程序, 必须处理属于另外一个段的数据, 比如它的段寄存器是 ES、FS 或 GS。例如一个大型的数据表, 它的数据是在程序的另一个段里从外部存储设备装入到存储器中的。

可以使用任何指令处理在其他段中的数据, 但必须确定合适的段寄存器。假定其他段的段地址在 ES 中, 并且 BX 的内容是该段内的偏移地址。要求从那个单元传送 2 个字节(一个字)到 DX:

MOV DX, ES:[BX] ; 从 ES:[BX] 传送到 DX

编码“ES:”代表跨越操作符,意思是“用ES代替正常使用的DS段寄存器”。下个例子是把一个字节值从CL传送到这个其他段,段的偏移值是由SI中的值加上36形成的:

MOV ES:[SI+36], CL ; 从 CL 传送到 ES:[SI+36]

汇编程序产生带有跨越操作符的目标码,该跨越操作符作为一个字节的前缀(26H)直接放在指令的前面,就好像已经编写成了2条指令

ES:MOV DX, [BX] ; 从 ES:[BX] 传送到 DX
ES:MOV [SI+36], CL ; 从 CL 传送到 ES:[SI+36]

6.9 近地址与远地址

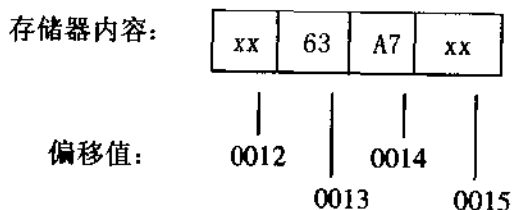
在程序中,地址可以是近的或远的。近地址仅仅是由地址的16位偏移部分组成。引用近地址的指令采用的是当前段,即对于数据段是DS,对于代码段是CS。

远地址是由段和偏移部分组成的32位的段:偏移的格式。指令可以引用来自当前段内的或是另一个段中的远地址。

几乎所有在实模式下的汇编程序设计都使用近地址,除非另有其它指定,近地址一般是由汇编程序产生的。由许多段组成的大型程序会需要远地址,因为该程序是用Flat存储模型来定义的。

6.10 对齐数据地址

由于8086和80286处理器有16位(字)数据总线,如果所存取的字从偶数(字)地址开始,那么它们的执行是比较快的。考虑这种情况,在偏移值0012H和0013H那里包含63A7H这个字。处理器可以在偏移值0012H处直接把整个字取到比如AX中。但该字可能开始于奇数地址,比如0013H:



在这种情况下,处理器不得不完成两次取数。第一次,在0012H和0013H处取2个字节并从0013H把字节(63)发送到AL。然后,在0014H和0015H处取2个字节并从0014H把字节(A7)发送到AH。现在AX的内容是A763H。

没有必要对偶数或奇数单元做任何特别的程序设计,也不一定要知道地址是偶数的还是奇数的。

80386+处理器有 32 位的数据总线, 因此建议按能被 4 整除的地址(双字地址)对齐所引用的项。(从技术上说, 486+处理器建议按 16 个字节(小段)边界对齐。)

可以使用 ALIGN 或 EVEN 伪操作强制按边界对齐项。例如, ALIGN 2 或 EVEN 可对齐字边界, 而 ALIGN 4 则是对齐双字边界。当汇编程序根据边界调整项的地址时, 位置计数器也要相应地增值。

由于用 PARA 定义的数据段是在小段边界上开始的, 所以可以首先用双字的值, 然后用字的值, 最后用字节的值来组织数据。对齐对于时间要求苛刻的应用程序可能是有帮助的。

6.11 要 点

- MOV 指令把按第二个操作数的地址引用的数据传送(或复制)到第一个操作数的地址中。
- LEA 指令对于用偏移地址去初始化寄存器是有用的。
- INC 和 DEC 是把寄存器或存储单元的内容加 1 和减 1。
- INT 指令中断程序的执行过程, 传送控制给 BIOS 或 DOS 去做指定的操作, 然后用 IRET 返回该程序并恢复处理过程。
- 操作数为指令提供数据的源。一条指令可以有 0 到 3 个操作数。
- 存在 2 个操作数时, 第二个操作数是源, 它引用立即数据或数据的地址(寄存器的或存储器的)。第一个操作数是目的, 它引用的是在寄存器或存储器中要被处理的数据。
- 在立即格式中, 两个操作数中的第二个含有常数值或表达式。立即操作数应当与目的的大小相匹配, 即两者都是字节、字或双字。
- 在直接存储器格式中, 一个操作数引用存储器单元, 而另一个操作数则引用寄存器。
- 间接寻址利用了处理器的段: 偏移寻址的能力。使用 BP、BX、DI 和 SI 寄存器, 编写在方括号内作为变址操作符。BP 与 SS 联系在一起, 成为 SS:BP, 用于处理堆栈中的数据。BX、DI 和 SI 是和 DS 联系在一起的, 分别成为 DS:BX, DS:DI 和 DS:SI, 用于处理数据段中的数据。
- 可以把寄存器组合成间接地址, 如[BX+DI], 它指的是 BX 中的偏移值加上 DI 中的偏移值。

6.12 习 题

6-1. 对于有 2 个操作数的指令, (a)哪个操作数是源?(b)哪个操作数是目的?

6-2. 对于以下每条不相关的指令, 写出目的操作数的结果。如果指令是无效的, 请指出来。假定 BYTE1 被定义为 DB 05。

指令	前	后
(a) MOV CX, 25H	: CX=0000H	CX=
(b) MOV CL, 0	: CX=FFFFH	CX=

```

(c) MOV  AX, BYTE1      ; AX=1234H      AX=
(d) ADD  DL, BYTE1      ; DX=0120H      DX=
(e) INC  DX              ; DX=FFFFH      DX=
(f) INC  DL              ; DX=FFFFH      DX=
(g) XCHG AH, AL         ; AX=1234H      AX=
(h) SUB  CX, CX          ; CX=1234H      CX=
(i) XCHG CX, CX         ; CX=1234H      CX=

```

6-3. (a)以下 MOV 指令在执行方面有什么明显的区别?

```

MOV  DX, AC24H
MOV  DX, [AC24H]

```

(b)对于第二条 MOV, 一个操作数是在方括号中的。该特征的名字是什么?

6-4. (a)以下 MOV 指令在执行方面有什么明显的区别?

```

MOV  BX, WORDA
MOV  [BX], WORDA

```

(b)对于第二条 MOV, 第一个操作数用的是哪种寻址方式?

6-5. 说明下列指令的操作

```

SUB  CX, [BX+DI+4]

```

6-6. 对于以下语句, (a)指出错误, (b)指出一种改正的方法:

```

SUB  [SI], [BX]

```

6-7. 给出以下数据定义, 找出语句中的错误, 并编写指令加以改正:

```

BYTE1 DB 48
BYTE2 DB 32
WORD3 DB 216
(a) MOV  BYTE1, BYTE2
(b) SUB  CL, WORD3      ; 第二个操作数是正确的
(c) ADD  DH, C51CH      ; 第二个操作数是正确的

```

6-8. 编写以下带立即操作数的指令:

(a)把 hex 26 加到 BX, (b)从 CX 减去 hex 26, (c)把 CH 右移 2 位, (d)把 BYTE1 左移 2 位, (e)把 426 存入 CX, (f)比较 BYTE1 和 hex 25。

6-9. 编写指令, 把名为 WORD3 的字的内容和 AX 交换。

6-10. 编写指令, 把名为 RATE_TBL 项的偏移地址放到 SI 中。

6-11. 用通用术语说明 INT 指令的用途。

6-12. 说明(a)INT 指令是如何影响堆栈的, (b)IRET 指令是如何影响堆栈的。

6-13. 编码, 汇编, 连接, 并使用 DEBUG 检查以下程序:

(a)定义名为 BYTE1 和 BYTE2(包含任何值)的字节项, 以及名为 WORD3(包含 0)的字项, (b)把 BYTE1 的内容传送到 AL, (c)把 BYTE2 的内容加到 AL, (d)把立即值 42H 传送到 DL, (e)交换 AL 与 DL 的内容, (f)AL 的内容乘以 DL(MUL DL), (g)把乘积从 AX 传送到 WORD3。

6-14. 说明为什么指令 ADD [BX], 25 是无效的, 并改正它。

6-15. 编写指令把 `BYTE_TBL` 中定义的每个字节逐次地加到 `CL` 寄存器中。使用直接-偏移值寻址和 5 个 `ADD` 指令。

```
BYTE_TBL DB 12, 15, 16, 10, 8
```

6-16. 利用 6-15 题所定义的 `BYTE_TBL` 编写指令向 `CL` 传送第 4 个字节(内容为 10)。(a) 使用间接寻址。(b)使用基址-位移量寻址。

6-17. 给出以下的表和初始化指令, 说明 `MOV` 的作用:

```
VALUE_TBL DB 1, 2, 3, 4, 5, 6, 7, 8
...
LEA BX, VALUE_TBL           ; 初始化 BX 和 DI
MOV DI, 4
(a) MOV CL, [BX]             ; CL=
(b) MOV DL, [BX+3]           ; DL=
(c) MOV AL, [BX+DI]          ; AL=
(d) MOV CH, 2[BX+DI]         ; CH=
```


目的：阐述对于程序控制(循环与转移)，逻辑比较，逻辑位操作，以及程序组织的要求。

7.1 引言

到本章为止，大多数程序都是直线执行的，即一条指令跟着另一条指令顺序地执行。但是，很少会有这种简单易编程序的问题。大多数程序是由各种测试和若干循环组成的。这些测试用来确定要采取几个动作中的哪一个动作，而在这些循环中的一系列步骤则要重复到指定的要求被满足为止。例如，通常的做法是去测试程序是否应该结束执行。

这些要求包括传送控制给一条指令的地址，而该指令不是紧跟在当前正在执行的指令之后。控制的传送可能是向前的，去执行一系列新的步骤；或者是向后的，去重新执行一些同样的步骤。能够传送控制的指令可以把控制传送到超出正常的顺序流程之外，这是通过改变IP中的偏移值的办法来完成的。

以下是在这一章里按类型列出的指令：

比较操作	传送操作	逻辑操作	移位和循环
CMP	CALL	AND	SAR/SHR
TEST	JMP	NOT	SAL/SHL
	Jnnn*	OR	RCR/ROR
	LOOP	XOR	RCL/ROL
	RETn		
* Jnnn 是指所有条件转移指令，如 JNE 和 JL。			

7.2 短地址，近地址和远地址

汇编程序支持三种类型的地址，是根据它们与当前地址的距离加以区别的：

- (1) 短地址, 对距离的限制是 -128(80H)到 127(7FH)个字节。
- (2) 近地址, 对距离的限制是 -32768(8000H)到 32767(7FFFH)个字节, 在同一个段内。
- (3) 远地址, 在同一段内, 距离可超过 32K, 或在另一个段里。

转移操作可到达 1 个字节偏移值的短地址, 还可到达一个或两个字偏移值的近地址。远地址则可到达段地址和偏移值所指定的地方。CALL 是为此目的所用的标准指令, 因为它便于连接到需要的地址并随后返回。

下面的表列出了 JMP, LOOP, 以及 CALL 操作在距离方面的规则。几乎没有必要记住这些规则, 因为正常使用这些指令难得出现问题。

	短	近	远
指令	-128 到 127 同一段	-32 768 到 32 767 同一段	超过 32K 或在 另一段
JMP	是	是	是
Jnnn	是	是 (80386+)	否
LOOP	是	否	否
CALL	不适用	是	是

JMP, Jnnn(条件转移), 或 LOOP 指令的操作数是另一条指令的标号。下面的例子是转移到 L10, L10 是 INC 指令的标号:

```

    JMP  L10
    ...
L10:  INC  CX
    ...

```

一条指令的标号, 如 L10:, 是用冒号来结束的, 这赋予它近的属性, 即此标号是在同一代码段的一个过程之内的。遗漏了冒号是个普通的错误, 汇编程序会发出信号。注意: 一条指令的操作数中的地址标号(比如 JMP L10)是没有冒号的。还可以把标号写在单独的行里, 如

```

L10:
    INC  CX

```

两种情况下, L10 的地址都是位于 INC 指令的第一个字节。

7.3 JMP 指令

通用的传送控制指令是 JMP(转移)指令。由于该操作在所有情况下传送控制, 所以这种转移是无条件的。JMP 还要填满处理器的预取指令队列, 这样有许多转移操作的程序可能会

降低处理速度。JMP 的格式是：

[label:]	JMP	short/near/far address
----------	-----	------------------------

(short: 短, near: 近, far: 远, address: 地址)

7.3.1 短转移与近转移

在同一段内的转移操作可以是短转移或近转移(或甚至是远转移, 如果目的是一个具有 FAR 属性的过程的话)。在汇编程序第一遍扫描源程序时, 会产生每条指令的长度。但是, JMP 指令可能是 2 个、3 个或 4 个字节长。JMP 操作到标号的距离在 -128(80H)到 127(7FH)字节范围之内的是短转移。汇编程序为操作产生一个字节(EB)并为操作数产生一个字节。操作数相当于偏移值, 当执行程序时, 处理器把它加到 IP 寄存器中。

超过 -128 到 +127 字节的 JMP 就变成了近转移(在 32K 范围内), 对于近转移, 汇编程序产生不同的机器码(E9)和 2 个字节的操作数(8086/80286)或 4 个字节的操作数(80386+). 暂时我们先不讨论远转移。

7.3.2 向后与向前转移

转移可以是向前的或是向后的。汇编程序已经遇见过这个范围在 -128 字节内的指定的操作数(向后转移), 比如:

```
L10:                ; 转移地址
...
JMP L10             ; 向后转移
```

这种情况下, 汇编程序产生一个 2 个字节的机器指令。在向前转移中, 汇编程序还没有遇到过这个指定的操作数:

```
JMP L20             ; 向前转移
...
L20:                ; 转移地址
```

汇编程序现在还不知道向前转移是短转移还是近转移, 某些版本假定是近转移并产生 3 个字节指令 EBxx90, 其中 xx 是偏移值, 而 90 是 NOP 的机器码。但是, 所提供的转移实际上是短转移, 可以使用 SHORT 操作符强制它是个短转移并产生 2 个字节指令 EBxx, 编码如下:

```
JMP SHORT L20
```

7.3.3 程序: 使用 JMP 指令

图 7-1 的程序说明 JMP 的用法。程序把 AX 与 BX 初始化为 0, 而把 CX 初始化为 1, 并且循环完成以下的操作:

- AX 加 1。
- 把 AX 加到 BX。

- 左移一位(CX 中的值加倍)。

在循环的末尾(SHL 之后), 指令 `JMP A20` 传送控制给标号为 A20 的指令。重复循环的结果使 AX 增加为 1, 2, 3, 4, ...; BX 按照数字的和增加为 1, 3, 6, 10, ...; 而 CX 则加倍为 1, 2, 4, 8, ...。由于这个循环没有退出, 所以处理过程是无休止的——通常仅有的适合的实际应用是用于像监控系统这样一类的专门用途。

在程序中, A20 距 `JMP` 是-9 个字节。可以用分析 `JMP` 的目标码(EBF7)的办法来确定这个距离。EB 是短 `JMP` 的机器码, 而 hex F7 则是-9 的二进制补码表示。因为这是一个向后转移, 所以操作数 F7 是负的。现在, IP 的内容是下一条要执行的指令的偏移值(0112H)。JMP 操作把 F7(由于 IP 的大小是一个字, 所以应转换为 FFF7。)加到 IP, IP 的内容是跟随 `JMP` 之后的指令的偏移值 0112H:

	十进制	十六进制
IP 寄存器:	274	0112
JMP 操作数:	-9	FFF7 (二进制补码)
<hr/>		
转移地址:	265	(1)0109

汇编程序计算的转移地址是 0109H(其中进位输出的 1 忽略不计)。检查一下, 这和程序列表中 A20 的偏移地址是一样的。该操作改变了 IP 中的偏移值, 填满了指令队列, 并继续重新执行跟在 A20 后面的指令。

	TITLE	A07JUMP (COM)	Using JMP for looping
		.MODEL SMALL	
		.CODE	
0100		ORG 100H	
0100	A10MAIN	PROC NEAR	
0100	B8 0000	MOV AX,00	; 把 AX 与 BX
0103	BB 0000	MOV BX,00	; 初始化为零,
0106	B9 0001	MOV CX,01	; CX 初始化为 01
0109	A20:		
0109	05 0001	ADD AX,01	; AX 加 1
010C	03 D8	ADD BX,AX	; AX 加到 BX 上
010E	D1 E1	SHL CX,1	; CX 加倍
0110	EB F7	JMP A20	; 重复
0112	A10MAIN	ENDP	
		END A10MAIN	

图 7-1 JMP 指令的用法

作为一种有用的做法, 可以使用 `DEBUG` 跟踪程序的多次迭代并观察 AX, BX, CX, 以及 IP 中的执行结果。完成 8 次迭代之后, AX 的内容是 08, BX 是 36, 而 CX 是 256(100H), 键入 Q 退出 `DEBUG`。

7.4 LOOP 指令

图 7-1 中所用的 `JMP` 指令产生一个无限的循环。标准的做法是编一个例行程序, 它循环指定的次数或直到满足指定的条件为止。`LOOP` 指令就是服务于这一目的的, 它要求在 CX 中有一个初始值。对于每次迭代, `LOOP` 自动地从 CX 中减 1。一旦 CX 达到零, 控制向下直

接到下一条指令；如果 CX 是非零，则控制转移到操作数地址。对操作数的距离必须是短转移，范围在-128 到+127 个字节。对于超出这一限制的操作，汇编程序会发出比如“相对转移超出范围”(relative jump out of range)的信息。LOOP 的格式是

[label:]	LOOP	short-address
----------	------	---------------

图 7-2 的程序说明 LOOP 的用法。它所完成的操作除了初始化 CX 为 8 且 8 次循环后结束之外，和图 7-1 的程序是一样的。由于 LOOP 要求使用 CX，所以这个程序用 DX 代替 CX，完成把初始值 1 加倍的工作。LOOP 指令取代了 JMP A20 指令，并且为了提高处理速度，INC AX(AX 加 1)取代了 ADD AX,01。

正如 JMP 那样，LOOP 的机器码操作数包含从该指令末尾到 A20 地址间的距离，当程序执行时，它被加到 IP 中。

	TITLE	A07100P (COM)	Illustration of LOOP
		.MODEL SMALL	
		.CODE	
0100		ORG 100H	
0100	A10MAIN	PROC NEAR	
0100	B8 0001	MOV AX,00	; 把 AX 与 BX
0103	BB 0001	MOV BX,00	; 初始化为零。
0106	BA 0001	MOV DX,01	; DX 初始化为 01
0109	B9 000A	MOV CX,8	; 初始化为
010C			; 8 个循环
010C	40	INC AX	; AX 加 1
010D	03 D8	ADD BX,AX	; AX 加到 BX 上
010F	D1 E2	SHL DX,1	; DX 加倍
0111	E2 F9	LOOP A20	; CX 减 1,
			; 若非零则循环
0113	B8 4C00	MOV AX,4C00H	; 结果处理
0116	CD 21	INT 21H	
0118	A10MAIN	ENDP	
		END A10MAIN	

图 7-2 使用 LOOP 指令

作为有用的练习，使用 DEBUG 跟踪整个 8 次循环的全过程并观察 AX、BX、CX、DX 和 IP 中的执行结果。当 CX 减到零时，AX、BX 和 DX 的内容分别是 0008H、0024H 和 0100H。

LOOP 指令有 2 个变种，它们也都是 CX 减 1。LOOPE/LOOPZ(当相等或为零时循环)只要 CX 不为零或者零条件被设置(ZF=1)便继续循环。LOOPNE/LOOPNZ(当不相等或不为零时循环)只要 CX 不为零或者零条件没有设置(ZF=0)便继续循环。

LOOP 和它的 LOOPxx 变种都不改变标志寄存器中的任何标志的设置。但是，由于在 LOOP 例行程序内的其他指令会改变标志，所以图 7-2 中使用的是 LOOP 而不是 LOOPxx 变种。

7.5 标志寄存器

在本章余下的内容中，需要有关标志寄存器的更为详细的知识。这个寄存器有 16 位，由各种指令设置来指明它们的操作结果。在所有情况下，标志要保留其设置，直到另一条指令把它改变为止。标志寄存器包含以下一些通用的位：

位编号:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
标志:								O	D	I	T	S	Z	A	P	C

下面从右到左讨论标志位。引用“数据项”是指数据在寄存器或存储器中。

(1) **CF(进位标志)**。包含一个数据项高阶(最左边)位的进位(0 或 1)输出,它是伴随无符号算术操作和某些移位与循环操作而产生的。一些磁盘操作使用 CF 去指明成功(0)或失败(1)。JC 和 JNC 测试这个标志。

(2) **PF(奇偶标志)**。包含的是算术操作后数据项的低阶 8 位的检查结果。若 1 的个数为偶数,则清除 PF 为 0;若 1 的个数为奇数,则把 PF 置成 1。不要把这个标志和第 1 章所讨论的奇偶位搞混淆了,在一般的程序设计中,很少会涉及到它。JP 和 JPO 测试这个标志。

(3) **AF(辅助进位标志)**。该标志和使用 ASCII 和 BCD 压缩字段的算术运算有关,这在第 13 章会涉及到。当 1 字节的算术操作造成位 3(从右边数第 4 位)的进位输出进到了位 4,此时 AF 位置 1。

(4) **ZF(零标志)**。根据算术操作或逻辑操作的结果进行清除或置位。非零的结果要把 ZF 清除为 0,而零的结果则要把 ZF 置成 1。但是,如果这种设置不是表面上的正确而是逻辑上的正确,那么“0”的意思是“不”(结果不等于零),而“1”的意思是“是”(结果等于零)。JE 和 JZ(连同其他指令一起)测试 ZF。

(5) **SF(符号标志)**。根据算术操作后的数据项符号(高阶位或最左边的位)设置:正值,把 SF 清除为 0;而负值则把 SF 置 1。JG 和 JL(连同其他指令一起)测试 SF。

(6) **TF(陷阱标志)**。当设置它时,处理器将按单步方式执行指令,即在用户的控制之下,每次执行一条指令。调试程序为单步执行而设置 TF,并且放在差不多是最适当的、你希望找到它的地方。INT 03 操作用于设置 TF。

(7) **IF(中断标志)**。当它为 0 时,禁止外部中断;为 1 时允许中断。IF 可以由 STI 指令设置,由 CLI 指令清除,通常用在关键性的场合。

(8) **DF(方向标志)**。由串操作在确定数据传送方向时使用。当 DF 为 0 时,串操作实行从左到右的数据传送;当 DF 为 1 时,串操作实行从右到左的数据传送。第 11 章会讨论到它。

(9) **OF(溢出标志)**。指明带符号算术操作后数据项的高阶(最左边)符号位的进位输出。JO 和 JNO 测试 OF。

7.6 CMP 指令

CMP 指令是用来比较两个数字数据字段的,寄存器中包含了一个或两个字段。CMP 的格式是

[label:]	CMP	register/memory, register/memory/immediate
----------	-----	--

从技术上说,可以使用 CMP 比较串(字符)数据,但是 CMPS(第 11 章会涉及到)才是实现这一目的合适指令。CMP 操作的结果影响 AF、CF、OF、PF、SF 和 ZF 标志,然而没有必要分别地去测试这些标志。以下代码是测试 DX 为零值的:

```
CMP DX, 00          ; DX=0?
```

```

        JE    L10                ; 如果是, 则转移到 L10
        .      (非零时的动作)
        .
L10:    ...                    ; DX=0 的转移点

```

如果 DX 是零, CMP 设置 ZF 为 1, 并且可能改变或不改变其他标志的设置。JE(若相等则转移)指令只测试 ZF。由于 ZF 是 1(意思是零状态), JE 传送控制(转移)给由操作数 L10 所指明的地址。

实际上, CMP 操作是比较第一个和第二个操作数。例如, 第一个操作数的值高于、等于或低于第二个操作数的值吗 (CMP 和 SUB 一样, 没有为了执行所需要的附加存储周期)? 下一节介绍基于测试条件进行传送控制的各种方法。

7.7 条件转移指令

处理器支持各种各样的条件转移指令, 这些指令根据标志寄存器的设置来传送控制。例如, 可以比较或相加 2 个字段, 然后根据比较所设置的标志值进行有条件的转移。条件转移的格式是

[label:]	Jnnn	short-address
----------	------	---------------

如前所述, LOOP 指令使 CX 减 1, 如果它是非零, 则控制传送给操作数地址。可以用 2 个语句取代图 7-2 中的 LOOP A20 语句, 这 2 个语句中的一个使 CX 减 1, 而另一个完成条件转移, 如下所示:

```

DEC    CX                ; 等效于 LOOP
JNZ    A20               ;
...

```

DEC 使 CX 减 1 并设置或清除零标志, 然后 JNZ 测试设置的零标志。如果 CX 是非零, 则控制转移到 A20; 而如果 CX 是零, 则控制往下到下一条指令(转移操作还要填满处理器的预取指令队列)。尽管 LOOP 使用受到限制, 但它执行的速度还是比较快的, 而且比用 DEC 和 JNZ 指令使用的字节数要少。

像 JMP 和 LOOP 那样, JNZ 的机器码操作数含有从指令的末尾到 A20 地址的距离, 操作把它加到 IP 寄存器。对于 8086/80286, 条件转移的距离必须是短的, 在-128 到+127 字节范围之内。如果操作超出这一限制, 汇编程序要发出“相对转移超出范围”的信号。80386+ 处理器提供了 32 位(近的)偏移值, 允许到达 32K 范围内的任意地址。

如果编写 386 操作符, 汇编程序会自动产生 4 字节的向前转移的机器码(例 1); 使用 SHORT 操作符可以产生 2 字节的机器码(例 2):

<p>例 1</p> <pre> .386 CMP BX, CX JE L20 </pre>	<p>例 2</p> <pre> .386 CMP BX, CX JE SHORT L20 </pre>
---	---

7.7.1 带符号与无符号数据

判别条件转移的用途, 应该了解它们的用法。所要实现的比较或算术操作的数据类型(无符号或带符号的)可以确定所使用的指令。无符号的数据项(逻辑数据)把所有的位作为数据位处理, 典型的例子如用户编号, 电话号码, 以及比率和系数这样一些数字值。带符号数据项(算术数据)把最左边的位当作符号来处理, 其中 0 表示正, 而 1 表示负, 典型的例子是数量, 银行结算以及温度等, 它们可能是正的, 可能是负的。

在下一个例子中, 假定 CX 是 11000110, DX 是 00010110。指令 CMP CX, DX 比较 CX 和 DX 的内容。如果把它们看成是无符号的数据, CX 是较大的; 如果把它们看成是带符号的数据, 则由于负的符号的原因, CX 是较小的。这里所用的 CMP 是有效的, 可以选择适当的条件转移指令, 比如 JB(低于则转移)用于无符号数据或 JL(小于则转移)用于带符号数据。

7.7.2 基于无符号(逻辑)数据的转移

以下条件转移用于无符号数据:

符 号	说 明	测试的标志
JE/JZ	相等则转移或为零则转移	ZF
JNE/JNZ	不相等则转移或不为零则转移	ZF
JA/JNBE	高于则转移或不低于/等于则转移	CF, ZF
JAE/JNB	高于/等于则转移或不低于则转移	CF
JB/JNAE	低于则转移或不高于/等于则转移	CF
JBE/JNA	低于/等于则转移或不高于则转移	AF, CF

这些条件转移中的每一个都可以用两个符号操作中的一个来表示, 应该选择比较清楚也比较能说明问题的那一个。

7.7.3 基于带符号(算术)数据的转移

以下条件转移是用于带符号数据的:

符 号	说 明	测试的标志
JE/JZ	相等则转移或为零则转移	ZF
JNE/JNZ	不相等则转移或不为零则转移	ZF
JG/JNLE	大于则转移或不小于/等于则转移	OF, SF, ZF
JGE/JNL	大于/等于则转移或不小于则转移	OF, SF
JL/JNGE	小于则转移或不大于/等于则转移	OF, SF
JLE/JNG	小于/等于则转移或不大于则转移	OF, SF, ZF

测试相等/为零(JE/JZ)和不相等/不为零(JNE/JNZ)的转移在无符号与带符号数据的两个表中都有, 这是因为条件的存在与有没有符号无关。

7.7.4 专用的算术运算测试

以下条件转移指令有专门的用途:

符 号	说 明	测试的标志
JCXZ	若 CX 为零则转移	没有
JC	进位为 1 则转移 (和 JB 一样)	CF
JNC	进位为 0 则转移	CF
JO	溢出则转移	OF
JNO	不溢出则转移	OF
JP/JPE	奇偶位为 1 则转移或奇偶性为偶则转移	PF
JNP/JPO	奇偶位为 0 则转移或奇偶性为奇则转移	PF
JS	符号位为 1 (负的) 则转移	SF
JNS	符号位为 0 (正的) 则转移	SF

JCXZ 测试 CX 的内容是否为零。这条指令不需要紧跟在算术或比较操作之后。JCXZ 的一种用法是可以放在一个循环的起点, 以保证如果 CX 最初为零, 就能绕过此例程序。JC 和 JNC 通常用于测试磁盘操作的成败。

不需要记住所有这些指令的名字或它们所测试的标志。但是, 作为一个提醒, 请注意无符号数据的转移是相等(equal), 高于(above)或低于(below); 反过来, 带符号数据的转移是相等(equal), 大于(greater)或小于(less)。转移对于测试进位, 溢出, 以及奇偶标志有独特的用途。汇编程序把符号翻译成目标码而不关心你所使用的是哪一条指令, 但是比如 JAE 和 JGE, 尽管表面上类似, 却测试不同的标志(因为 JAE 适用于无符号数据, 而 JGE 则适用于带符号数据)。

80386+处理器允许远条件转移。可以指定短转移和远转移, 例如:

```
JNB    SHORT address
JBE    FAR address
```

7.7.5 测试多个条件

对于条件转移, 真的条件可以产生转移; 相反若是假的条件, 就直接往下执行下一条指令。程序可能必须测试一系列有关条件。在下面, 例 1 确定任一条件是否为真(OR 操作), 而例 2 确定所有条件是否为真(AND 操作):

例 1 任一条件为真

```
CMP AL, BL
JE equal
CMP AL, BH
JE equal
```

例 2 所有条件为真

```
CMP AL, BL
JNE not_equal
CMP AL, BH
JNE not_equal
```

CMP AL, CL	CMP AL, CL
JE equal	JNE not_equal
not_equal: <processing>	equal: <processing>
...	...
equal: <processing> ...	not_equal: <processing>

在例 1 中, 真的条件是按相等测试的, 并且所有转移都到 `equal` 标号; 在例 2 中, 真的条件是按不相等测试的, 并且转移到 `not_equal` 标号。实际上, 例 2(AND 操作)是使测试(JNE)和转移点(not_equal)都反向了。

7.8 调用过程

到现在为止, 例子中的代码段只包括一个过程, 编码如下:

```
proc-name PROC FAR
...
proc-name ENDP
```

在这种情况下的 `FAR` 操作数通知汇编程序和连接程序所定义的过程名是程序执行的入口点, 而 `ENDP` 伪操作则定义过程的结束。但是, 代码段可能包括任意个过程, 每一个过程是用它自己的 `PROC` 和 `ENDP` 伪操作来区分的。被调用的过程(或子程序)是完成有明确定义的任务的一组代码(比如设置光标或取得键盘输入)。把一个程序组织成过程能提供以下好处:

- 减少代码的数量, 因为一个公共的过程可以在代码段的许多地方被调用;
- 有助于更好的程序组织;
- 便于程序的调试, 因为可以更清楚地找出缺陷;
- 有助于程序的维护, 因为过程容易判定要做的修改。

程序设计的习惯是在每个过程的起点提供一些注释, 以便确定过程的用途和使用的寄存器, 并且如有必要, 在开始的时候使要改变的寄存器进栈, 而在退出时使它们出栈。

CALL 与 RETn 操作

`CALL` 指令的目的是把控制传送给被调用的过程(在本书中转移到过程仅有的例子是在 `COM` 程序的开始处)。`RETn` 指令是和 `CALL` 指令配对的, 作用是从被调用的过程返回到原先的调用过程。`RETn` 通常是被调用的过程中的最后一条指令。`CALL` 和 `RETn` 的格式是:

[label:]	CALL	procedure-name
[label:]	RET[n]	immediate

(procedure: 过程, name: 名)

汇编程序能区分过程的 `RET` 是近的还是远的并产生相应的目标码。然而, 为了清楚起见, 可以为近返回编写 `RETN`, 为远返回编写 `RETF`。`CALL` 和 `RET` 产生特定的目标码, 这取决于操作包含 `NEAR` 还是 `FAR` 过程。

(1) 近调用与返回。在同一段内调用(`CALL`)一个过程是近调用, 其实现如下:

- 通过进栈操作, `SP` 减 2(一个字)并把 `IP`(含有跟在 `CALL` 之后指令的偏移值)传送给进栈。

- 把被调用过程的偏移地址放入 IP 中(而且要填满处理器的预取指令队列)。

RET(或 RETN)从一个近过程返回,基本上是 CALL 的相反步骤:

- 老的 IP 值从堆栈出栈,回送到 IP(而且要填满处理器的预取指令队列)。
- SP 加 2。

现在 CS: IP 指向调用过程中跟在原先的 CALL 后的指令,并且在那里恢复执行。

(2) 远调用与返回。远调用 (CALL) 去调用标明 FAR 的过程,这个过程可能在另外一个代码段里。远调用使 CS 和 IP 都进栈,并且 RET(或 RETF)使它们都出栈。远调用与返回在第 22 章讨论。

(3) 近调用与返回举例。使用近调用与返回的典型程序组织如图 7-3。注意以下特点:

- 程序分为一个远过程 A10MAIN 和 2 个近过程 B10 与 C10。每个过程有唯一的名字,并有自己的 ENDP 用来结束其定义。
- A10MAIN 的 PROC 伪操作有 FAR 属性,因为它是操作系统中的入口点。
- B10 和 C10 的 PROC 伪操作有 NEAR 属性,它指明这些过程是在当前代码段范围内的。因为省略属性汇编程序默认为 NEAR,所以许多后面的例子都省略了它。
- 在过程 A10MAIN 中, CALL 传送程序控制给过程 B10 并开始它的执行。
- 在过程 B10 中, CALL 传送程序控制给过程 C10 并开始它的执行。
- 在过程 C10 中, RET 使控制返回到紧跟在 CALL C10 之后的指令。
- 在过程 B10 中, RET 使控制返回到紧跟在 CALL B10 之后的指令。
- 然后,过程 A10MAIN 从那一点起恢复处理过程。
- RET 总是返回到调用它的例程序。如果 B10 不是用 RET 结束,则处理过程会继续经过 B10 并直接往下到 C10。实际上,如果 C10 没有用 RET 结束,该程序会经过 C10 终点继续执行指令(如果有的话,可能是碰巧在那里的任何指令),结果是不可预见的。

		TITLE A07CALLP (EXE) Calling procedures		
		.MODEL SMALL		
		.STACK 64		
		.DATA		
		;-----		
		.CODE		
0000		A10MAIN	PROC	FAR
0000	E8 0006 R	CALL	B10	; 调用 B10
		; ...		
0003	B8 4C00	MOV	AX, 4C00H	; 结束处理
0006	CD 21	INT	21H	
0008		A10MAIN	ENDP	
		;-----		
0008		B10	PROC	NEAR
0008	E8 000C R	CALL	C10	; 调用 C10
		; ...		
000B	C3	RET		; 返回到调用程序
000C		B10	ENDP	
		;-----		
000C		C10	PROC	NEAR
		; ...		
000C	C3	RET		; 返回到调用程序
000D		C10	ENDP	
		;-----		
		END	A10MAIN	

图 7-3 调用过程

如前所述,可以用常规的排成行的代码传送控制给一个近过程,还可以用转移指令进入

一个近过程。但是，为了清晰和一致起见，习惯的做法是使用 CALL 传送控制给一个过程，并且使用 RET/RETN 结束过程的执行。

7.9 程序执行对堆栈的影响

到现在为止，程序几乎不需要把数据放入堆栈，因此只需要定义一个非常小的堆栈。但是正如图 7-3 所说明的，一个被调用的过程可以调用另外一个过程，而这个过程又可以调用另一个过程，所以堆栈必须足够大，以便容纳所有进栈的地址。弄清楚所有这些结果比它第一次出现要容易一些，并且定义一个 32 个字的堆栈对我们的大多数用途来说是很宽裕的。

CALL 和 PUSH 都是把一个字的地址或值存入堆栈。RET 和 POP 则是出栈和取出以前进栈的字。所有这些操作都是为下一个字而使 SP 寄存器中的偏移地址增量或减量。由于这一特点，RET 和 POP 必须分别和它们相应的 CALL 与 PUSH 操作相匹配。

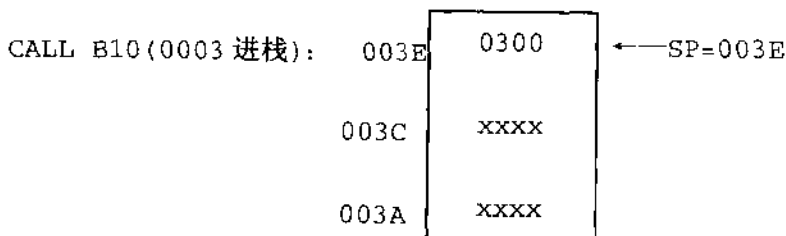
在为执行而装入 EXE 程序时，装入程序要初始化以下寄存器的值：

- DS 与 ES: PSP 的地址，放在存储器中的可执行模块之前的一个 256 个字节(100H)的区域。
- CS: 代码段的地址，程序的入口点。
- IP: 如果第一条可执行指令是在代码段起点，则为零。
- SS: 堆栈段的地址。
- SP: 栈顶的偏移值。例如，堆栈定义为 STACK 64(64 个字节或 32 个字)，SP 初始内容是 64 或 40H。

跟踪图 7-3 中简单程序的整个执行过程。实际上，被调用的过程可以包含任意多的指令。你会发现汇编这个程序并使用调试程序跟踪它的执行过程，同时检查 IP、SP 和堆栈的内容，所有这些都是值得一做的事。

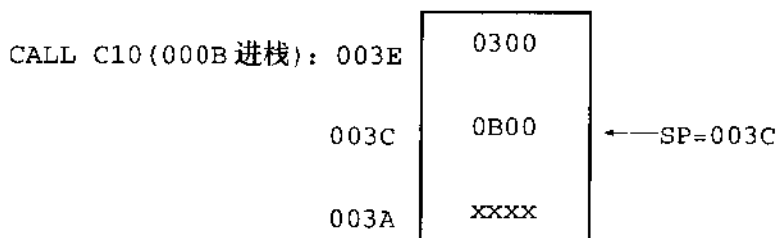
对于进栈来说，当前可用的单元是栈顶。在这个例子中，装入程序要把 SP 设置成堆栈的大小——64(40H)个字节。存储器中的字包含按相反顺序排列的字节，例如，0003 变成 0300。该程序完成以下操作：

- CALL B10 使 SP 减 2，从 40H 变成 3EH。然后使 IP(内容为 0003，下一条指令的地址)进栈，放入偏移值 3EH 处的栈顶。处理器使用由 CS: IP 形成的地址把控制传送到 B10，如以下堆栈帧所示：

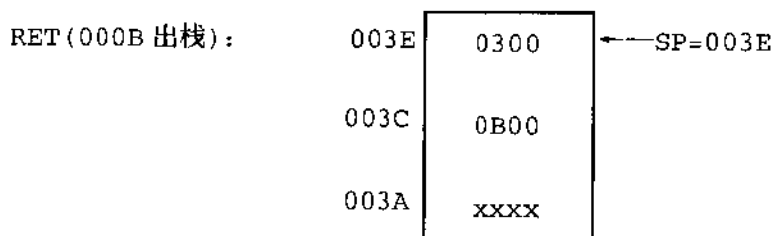


- 在过程 B10 中，CALL C10 使 SP 减 2，变成 3CH。然后使 IP(内容是 000B)进栈，放

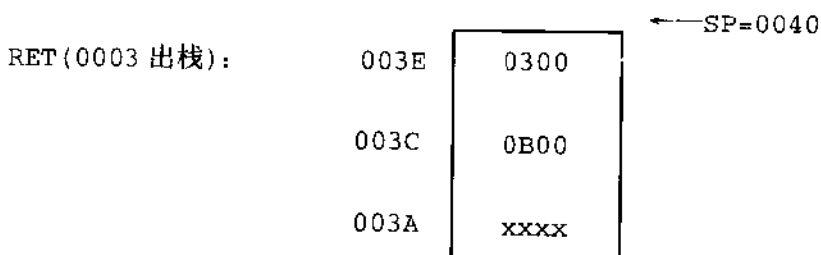
入偏移值 3CH 处的栈顶。处理器使用 CS: IP 地址把控制传送给 C10:



- 为了从 C10 返回, RET 使偏移值(000B)从 3CH 处的栈顶出栈, 放入 IP 中, 并且 SP 加 2 成为 3EH。IP 中的偏移值使之自动返回到过程 B10 中的 000BH 偏移值处:



- 在过程 B10 末尾的 RET 使地址(0003) 从栈顶 003EH 处出栈, 放入 IP 中, 并且 SP 加 2 成为 0040H。IP 中的偏移值使之自动返回到偏移值 0003H 处, 在那里程序结束执行。



如果利用调试程序查看堆栈, 可以发现由以前执行过的程序留在那里的未被破坏的数据。

过满的堆栈。考虑仅用 12 个字定义堆栈的程序。当执行时, 一连串的 CALL 和 PUSH 操作会从存储器高端到低端填满堆栈。此刻, SP=0。由于没有 RET 或 POP, 所以下一个 CALL 或 PUSH 使 SP 减量, 从 0 到 FFEH 并存放新的值, 这已超过所定义的堆栈, 会产生不可预见的后果。

传送参数

当调用过程时, 通常的做法是要传送一些参数(或自变量)供过程使用。程序可以通过值(实

际的数据项), 或者通过引用(数据的地址)传送参数。同样, 还可以用寄存器或用堆栈来传送参数。

1. 通过值传送参数。下面的例 1 和例 2 说明通过值传送参数。程序把被乘数与乘数作为参数传送给一个过程, 使它们简单地相乘。这个版本的 MUL 假定被乘数是在 AX 中, 而乘数是在操作数中, 求出来的乘积在 DX:AX 中。

例 1 传送在寄存器中的值

```

MOV AX, MULTICAND
MOV BX, MULTIPLIER
CALL M30MULT
...
M30MULT PROC NEAR
MUL BX
RET
M30MULT ENDP

```

例 2 传送在堆栈中的值

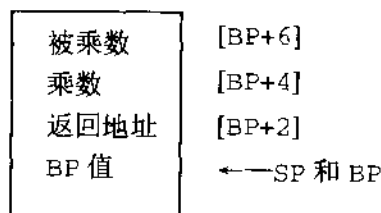
```

PUSH MULTICAND
PUSH MULTIPLIER
CALL M30MULT
...
M30MULT PROC NEAR
PUSH BP
MOV BP, SP
MOV AX, [BP+6]
MUL WORD PTR [BP+4]
POP BP
RET 4
M30MULT ENDP

```

例 2 值得进一步研究。在 M30MULT 的入口点上, SP 指向在堆栈中的返回地址。为了访问堆栈, 需要间接寻址, SP 做不到, 但 BP 可以做到。该过程通过以下方式实现:

- BP 进栈, 以便保留其内容。堆栈帧现在是这样:



- 复制 SP 偏移值到 BP 中。
- 使用 BP 作间接寻址去取在 [BP+6] 和 [BP+4] 中的已传送来的参数。
- BP 出栈(该操作还要使 SP 加 2 指向返回地址)。
- 执行 RET 4, 实现 2 个功能:

(1) 把返回地址装入 IP 中并使 SP 加 2(现在它指向堆栈中的乘数)。

(2) 把 RET 立即值(或出栈值)4 加到 SP, 实际上是从堆栈中“移出”2 个参数。注意, 由于被调用的过程没有使参数出栈, 所以 RET 必须调整 SP。

2. 通过引用传送参数。下面的例 3 和例 4 说明通过引用传送参数, 这也是高级语言调用了程序的方法。被传送的参数是乘数与被乘数的地址。

例 3 寄存器中的地址

例 4 堆栈中的地址

```

        LEA BX, MULTICAND          .386 ; 以下2个 PUSH 所需要的
        LEA SI, MULTIPLIER        PUSH  OFFSET MULTICAND
        CALL M30MULT              PUSH  OFFSET MULTIPLIER
        ...                       CALL  M30MULT
        ...                       ...
M30MULT PROC NEAR
        MOV AX, [BX]              M30MULT PROC NEAR
        MUL WORD PTR [SI]         PUSH  BP
        RET                      MOV  BP, SP
M30MULT ENDP                    MOV  BX, {BP+6}
                                MOV  DI, {BP+4}
                                MOV  AX, {BX}
                                MUL  WORD PTR [DI]
                                POP  BP
                                RET   4
                                M30MULT ENDP

```

用调试程序查看这些例子是如何工作的。

传送参数方法的选择取决于情况和习惯。例如，通过值传送的工作只是对于一些小的值；把数组作为参数传送时就需要通过引用来完成。同样，高级语言是通过引用来传送参数的。

被调用过程的通常做法是必须使所用的一个或多个寄存器进栈，并且在返回之前使它们出栈。当程序使用堆栈传送参数时，被调用的过程是在把 SP 传送到 BP 之后再使寄存器进栈的：

```

        PUSH BP                    ; BP 进栈
        MOV BP, SP
        PUSHF                      ; 标志进栈和
        PUSHA                     ; 所有寄存器进栈
        ...
        POPA                      ; 所有寄存器，
        POPF                      ; 标志，
        POP BP                    ; 和 BP 出栈

```

第22章会阐述汇编子程序的参数传送问题。

7.10 布尔操作

布尔逻辑在电路设计和并行的程序逻辑设计方面是很重要的。布尔逻辑的指令包括 AND, OR, XOR, TEST, 以及 NOT。它们全都可以用于清除，设置，以及测试各位。布尔操作的格式是

[label:]	operation	register/memory, register/memory/immediate
----------	-----------	--

第一个操作数引用寄存器或存储器中的一个字节、字或双字，而且是唯一变化的值。第二个操作数引用寄存器、存储器或立即值，但存储器到存储器的操作是无效的。该操作使2个引用的操作数的各位相对比，并且相应地设置 CF、OF、PF、SF 和 ZF 标志(AF 是未定义的)。

需要记住这些有用的规则：和 0 相“与”(AND)的位被清除为 0，而和 1 相“或”(OR)的位则被置成 1。

1. AND 指令。在 AND 情况下，如果相对比的位是 2 个 1，那么操作将结果置成 1，所有其他情况结果均为 0：

```
操作数 1:          0101
AND 操作数 2:      0011
-----
在操作数 1 中的结果: 0001
```

对于下面各不相关的例子，假定 BL 的内容是 0011 1010，CH 的内容是 1010 0011：

```
(1) AND BL, 0FH          ; 把 BL 置成 0000 1010
(2) AND BL, 00H          ; 把 BL 置成 0000 0000
(3) AND BL, CH           ; 把 BL 置成 0010 0010
```

例(2)提供了一种把寄存器清除为 0 的方法。

2. OR 指令。在 OR 情况下，如果其中一个(或两个)相对比的位是 1，则操作将置结果为 1；如果两个位都是 0，则结果是 0：

```
操作数 1:          0101
OR 操作数 2:       0011
-----
在操作数 1 中的结果: 0111
```

对于以下不相关的例子，假定 BL 的内容是 0011 1010，CH 的内容是 1010 0011：

```
(1) OR CH, BL          ; 把 CH 置成 1011 1011
(2) OR CH, CH          ; 置 SF 和 ZF
```

虽然使用 CMP 可能更清楚些，但还是可以为以下目的使用 OR：

```
(1) OR DX, DX          ; 测试 DX,
    JZ exit            ; 如为零则转移
(2) OR DX, DX          ; 测试 DX,
    JS exit            ; 如为负则转移
```

3. XOR 指令。在 XOR 的情况下，如果相对比的位不相同，则操作置结果为 1。如果相对比的位是一样的(都是 0 或都是 1)，则结果是 0。

```
操作数 1:          0101
XOR 操作数 2:      0011
-----
在操作数 1 中的结果: 0110
```

对于以下不相关的例子，假定 BL 的内容是 0011 1010，CH 的内容是 1010 0011：

```
(1) XOR BL, 0FFH       ; 把 BL 置成 1100 0101
(2) XOR BL, BL         ; 把 BL 置成 0000 0000
```

例(2)提供把寄存器清除为零的另一种方法。

4. TEST 指令。TEST 和 AND 设置标志的做法相同，但不改变目的操作数中所引用的位。如果任何相对比的位两个都为 1，则 TEST 清除零标志。下面是几个例子：

```
(1) TEST CX, 0FFH      ; 测试 CX 的内容
```



```

JZ    exit          ; 是零值吗?
(2) TEST BL, 00000001B ; 测试 BL 的内容
JNZ    exit          ; 是奇数吗?
(3) TEST CL, 11110000B ; CL 的最左边 4 位中有
JNZ    exit          ; 非零位吗?

```

5. NOT 指令。NOT 把寄存器或存储器中的字节、字或双字中的各位简单地变反，即 0 变成 1，1 变成 0。实际上，就是二进制反码。NOT 的格式是：

[label:]	NOT	register/memory
----------	-----	-----------------

例如，如果 BL 的内容是 0011 1010，则指令 NOT BL 把 BL 变成 1100 0101(其效果正好和先前例子中的 XOR BL, 0FFH 相同)，标志位不受影响。

由此可见，NOT 和 NEG 是不同的：NEG 实现的是二进制补码，它通过按位求反并加 1 把一个二进制值从正的变为负的，反过来一样。通常，NOT 用于无符号数据，而 NEG 则用于带符号数据。

程序：大写与小写字母间的转换

由于多方面的原因，大写字母与小写字母之间要进行转换。例如，你可能有个数据文件，在这个文件里，所有的字母数据都是大写字母。或者，一个程序允许用户用大写字母或小写字母(如“YES”或“yes”)输入值，然后转换成大写字母，以便于对它进行测试。大写字母 A 到 Z 用 ASCII 值 41H 到 5AH 表示，而小写字母 a 到 z 则用 61H 到 7AH 表示。唯一的区别在于：对于大写字母位 5 是 0，而对于小写字母位 5 是 1，如下所示：

TITLE	A07CASE (COM)	Change uppercase to lowercase
	.MODEL SMALL	
	.CODE	
	ORG 100H	
BEGIN:	JMP A10MAIN	

CONAME	DB 'LASER-12 SYSTEMS', '\$'	

A10MAIN	PROC NEAR	
	LEA BX, CONAME+1	; 要改变的第一个字符
	MOV CX, 15	; 要改变的字符数
A20:		
	MOV AH, [BX]	; 从 CONAME 取得的字符
	CMP AH, 41H	; 是
	JB A30	; 大
	CMP AH, 5AH	; 写
	JA A30	; 字母吗?
	XOR AH, 00100000B	; 是，转换
	MOV [BX], AH	; 送回 CONAME
A30:		
	INC BX	; 为下一字符而设置
	LOOP A20	; 循环 15 次
		; 完成。
	MOV AH, 09H	; 显示
	LEA DX, CONAME	; CONAME
	INT 21H	
	MOV AX, 4C00H	; 结束处理
	INT 21H	
A10MAIN	ENDP	
	END BEGIN	

图 7-4 把大写字母改变成小写字母

	大写		小写
字母 A:	01000001	字母 a:	01100001
字母 Z:	01011010	字母 z:	01111010
位:	6543210	位:	76543210

图 7-4 的程序从 CONAME+1 处开始对数据项 CONAME 的内容进行从大写 to 小写的转换。该程序用 CONAME+1 的地址初始化 BX，并且使用该地址从 CONAME+1 开始向 AH 传送每个字符。如果该值是在 41H 和 5AH 之间，那么 XOR 指令就把位 5 清除为 0:

```
XOR AH, 00100000B
```

除了 A 到 Z 外，所有字符都保持不变。然后该程序把改变了的字符送回到 CONAME，并为处理下一个字符而使 BX 加 1。程序循环 15 次，从 CONAME+1 开始每个字符循环一次。这种方法的使用使 BX 起到寻址存储单元的变址寄存器的作用。为了同样目的，可以使用 SI 和 DI。最后，程序显示改变了的 CONAME 的内容。

7.11 移 位

移位指令是计算机逻辑功能的一部分，可以完成以下操作：

- 引用寄存器或存储器地址。
- 左移或右移。
- 一个字节最多移 8 位，一个字最多移 16 位，一个双字最多移 32 位。
- 逻辑移位(无符号)和算术移位(带符号)。

7.11.1 SHR/SAR/SHRD: 向右移位

SHR(逻辑右移)，SAR(算术右移)，以及 SHRD(双字右移)操作是在指定的寄存器或存储单元中向右移位。SHR 和 SAR 的格式是：

[label:]	SHR/SAR	register/memory, CL/immediate
----------	---------	-------------------------------

第二个操作数包含移位值，该值是个立即数或是对 CL 寄存器的引用。对于 8088/8086 处理器而言，立即数移位值只能是 1，更大的值必须放在 CL 中，后继的处理器允许立即移位值最大达到 31。

每个移出的位放入进位标志中。SHR(逻辑右移)是为逻辑(无符号的)数据提供的，而 SAR(算术右移)则是为算术(带符号的)数据提供的：

SHR: 0 →

--	--	--	--	--	--	--	--

 C

SHR: S →

--	--	--	--	--	--	--	--

 C

SHRD 会在下一节说明。以下相关的指令说明使用 SHR 将无符号数据移位：

指令	注释	二进制	十进制	CF
MOV BH, 10110111B	; 初始化 BH	10110111	183	
SHR BH, 01	; 右移 1 位	01011011	91	1
MOV CL, 02	; 置移位值			
SHR BH, CL	; 再右移 2 位	00010110	22	1
SHR BH, 02	; 再右移 2 位	00000101	5	1

第一条 SHR 把 BH 的内容向右移一位。现在移出的 1 驻留在进位标志中，而 0 填充到 BH 的左边。第二条 SHR 使 BH 再移两位。进位标志包含相继的 1 和 1，而两个 0 填充到 BH 的左边。第三个 SHR 也是使 BH 再移了两位。

SAR 不同于 SHR 的一个重要方面是：SAR 是用符号位去填充最左边腾空的位。这样，正值和负值都保留它们的符号。以下相关的指令说明使用 SAR 将带符号的数据移位：

指令	注释	二进制	十进制	CF
MOV BH, 10110111B	; 初始化 BH	10110111	-73	
SAR BH, 01	; 右移 1 位	11011011	-37	1
MOV CL, 02	; 置移位值			
SAR BH, CL	; 再右移 2 位	11110110	-10	1
SAR BH, 02	; 再右移 2 位	11111101	-3	1

对于二等分值，右移是特别有用的，而且在有效的执行速度上要比用除法操作更快。在 SHR 和 SAR 的例子中，第一次右移 1 位相当于除以 2，而第二次和第三次右移两位，每次都相当于除以 4。

二等分奇数，比如 5 和 7 会分别产生 2 和 3，并把进位标志置成 1。在移位操作之后，可以用 JC(如进位则转移)测试进位标志的状态(0 或 1)。

在 32 位寄存器中右移。考虑一个 32 位的值，它最左边的 16 位在 DX 中，而最右边的 16 位在 AX 中，如同 DX:AX。下面的例子把 DX:AX 传送给 32 位的 ECX 寄存器，其中右移操作将该值除以 2：

```
MOV CX, DX           ; DX 送 ECX 的低半部分
SHL ECX, 16          ; 移位到 ECX 的高半部分
MOV CX, AX           ; AX 送 ECX 的低半部分
SHR ECX, 01          ; ECX 除以 2
```

对于 80386 及其后继机型，SHRD 可以用于移 16 位或 32 位值。它的格式是：

[label:]	SHRD	register/memory, register, CL/immediate
----------	------	---

第一个操作数接收被移出的位。第二个操作数的长度和第一个操作数相同，它的内容是要被移动的所有位。第三个操作数(CL 或立即值)的内容是移位计数值。下面是几个例子：

```
SHRD CX, DX, 4       ; 从 DX 到 CX 右移 4 位
SHRD ECX, EBX, CL    ; 从 EBX 到 ECX 右移?位
```

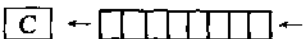
7.11.2 SHL/SAL/SHLR：向左移位

SHL(逻辑左移)，SAL(算术左移)，以及 SHLD(双字左移)操作是在指定的寄存器或存储单元中向左移位。它们的格式是：

[label:]	SHL/SAL	register/memory, CL/immediate
----------	---------	-------------------------------

第二个操作数包含移位值，它是立即值或是对 CL 寄存器的引用。对于 8088/8086 处理器，立即值只可以是 1，更大的值必须放在 CL 中，而其后继的处理器允许立即移位值最大达到 31。

每个移出的位进入进位标志。SHL 和 SAL 在操作方面是相同的，而且它们都可以左移逻辑(无符号)和算术(带符号)数据，也就是说，左移无符号数据和左移带符号数据是没有区别的：

SHL 和 SAL: 

SHLD 将在下一节说明。以下相关的指令说明用 SHL 移位无符号数据：

指令	注释	二进制	十进制	CF
MOV BH, 00000101B	; 初始化 BH	00000101	5	
SHL BH, 01	; 左移 1 位	0001010	10	0
MOV CL, 02	; 置移位值			
SHL BH, CL	; 再左移 2 位	00101000	40	0
SHL BH, 02	; 再左移 2 位	10100000	160	0

第一条 SHL 把 BH 的内容向左移一位。移出的 1 现在驻留在进位标志中，而把 0 填入 BH 的右边。第二条 SHL 再使 BH 移两位。进位标志包含的是相继的 0 和 0，而两个 0 填入 BH 的右边。第三条 SHL 再使 BH 移两位。

左移总是向右填充 0。SHL 和 SAL 的结果是一样的，以至于如果 SAL 用在前面的例子中，效果相同。对于值的加倍，左移是特别有用的，而且比用乘法操作在执行时间上明显地要快一些。在左移操作的例子中，第一次左移 1 位，相当于乘以 2，而第二次和第三次左移两位，相当于乘以 4。

移位操作之后，可以使用 JC(如进位则转移)指令测试移入进位标志的位。

在 32 位寄存器中左移。考虑一个 32 位的值，它的最左边的 16 位在 DX 中，而最右边的 16 位在 AX 中，如同 DX:AX。下面的例子是把 DX:AX 传送到 ECX，其中的移位操作加倍该值：

MOV CX, DX	; DX 送 ECX 的低半部分
SHL ECX, 16	; 移位到 ECX 的高半部分
MOV CX, AX	; AX 送 ECX 的低半部分
SHL ECX, 01	; ECX 乘以 2

对于 80386 和后继的型号，SHLD 可以用于 16 位和 32 位值的左移。它的格式是：

[label:]	SHLD	register/memory, register, CL/immediate
----------	------	---

第一个操作数接收被移出的位。第二个操作数的长度和第一个操作数相同，它的内容是要被移位的所有位。第三个操作数(CL 或立即值)包含移位计数值。下面是几个例子：

SHLD BX, DX, 2	; 从 DX 到 BX 左移 2 位
SHLD EAX, EDX, CL	; 从 EDX 到 EAX 左移?位

7.12 循环移位

循环指令是计算机逻辑能力的一部分，可以完成以下操作：

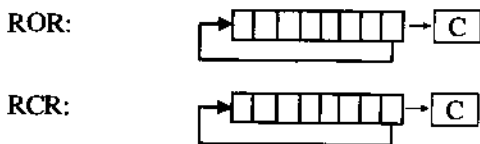
- 引用寄存器或存储器。
- 循环右移或循环左移。被循环移出的位填充寄存器或存储单元中被腾空的位，还可以复制到进位标志中。
- 一个字节最多循环 8 位，一个字最多循环 16 位，一个双字最多循环 32 位。
- 逻辑(无符号)循环移位或算术(带符号)循环移位。

第二个操作数包含循环值，它是常数(立即值)或是对 CL 寄存器的引用。对于 8088/8086 处理器，循环的值只能是 1，大于 1 的值必须放在 CL 中，后继的处理器允许立即循环值最大达到 31。循环指令的格式如下：

[label:]	rotate	register/memory, CL/immediate
----------	--------	-------------------------------

7.12.1 ROR/RCR：循环右移

ROR 和 RCR 操作在指定的寄存器或存储单元中向右循环移位。移出的每一位进入进位标志中。ROR(逻辑循环右移)是为逻辑(无符号的)数据提供的，而 RCR(带进位循环右移)则是为算术(带符号的)数据提供的：



以下相关的指令说明 ROR 的用法：

指令	注释	二进制	CF
MOV BL, 10110111B	；初始化 BL	10110111	-
ROR BL, 01	；循环右移 1 位	11011011	1
MOV CL, 03	；置移位值		
ROR BL, CL	；再循环右移 3 位	01111011	0
ROR BL, 03	；再循环右移 3 位	01101111	0

第一条 ROR 把 BL 最右边的 1 循环移到最左边已被腾空的位置并进入 CF。第二和第三条 ROR 操作是把最右边的 3 位循环移到最左边已被腾空的位置并进入 CF。

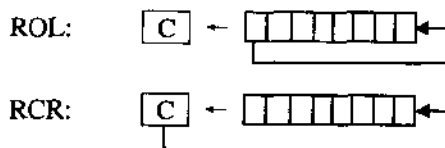
下面的 RCR 指令在 ECX 中移位：

RCR ECX, 6 ；循环右移双字的 6 位

由此可见，RCR 和 ROR 的区别是：RCR 向右循环移出的每一位首先传送到 CF，而 CF 位传送到左边已被腾空的位置。

7.12.2 ROL/RCL: 循环左移

ROL 和 RCL 操作是在指定的寄存器和存储单元中向左循环移位。移出的每一位进入进位标志中。ROL(逻辑循环左移)是为逻辑(无符号的)数据提供的,而 RCL(带进位循环左移)则是为算术(带符号的)数据提供的:



以下相关指令说明 ROL 的用法:

指令	注释	二进制	CF
MOV BL, 10110111B	; 初始化 BL	10110111	-
ROL BL, 01	; 循环左移 1 位	01101111	1
MOV CL, 03	; 置移位值		
ROL BL, CL	; 再循环左移 3 位	01111011	1
ROL BL, 03	; 再循环左移 3 位	11011011	1

第一条 ROL 把 BL 最左边的 1 循环移到最右边已被腾空的位置并进入 CF。第二和第三条 ROL 操作是把最左边的 3 位循环移到最右边已被腾空的位置并进入 CF。

下面的 RCL 指令在 EBX 中移位:

RCL EBX, 4 ; 循环左移双字的 4 位

由此可见, RCL 和 ROL 的区别是: RCL 向左循环移出的每一位首先传送到 CF, 然后 CF 位传送到右边已被腾空的位置。在 RCR 或 RCL 操作之后, 可以使用 JC(如进位则转移)指令测试循环移入 CF 的位。

7.12.3 双字移位与循环

可以使用用于乘除的移位与循环指令进行双字乘以 2 的操作。考虑一个 32 位值, 它的最左边 16 位在 DX 中, 而最右边的 16 位在 AX 中, 形成 DX:AX。把该值“乘”以 2 的指令是:

SHL AX, 1 ; 使用左移将
RCL DX, 1 ; DX: AX 乘以 2

SHL 操作把 AX 中的所有位向左移, 并且最左边的位移入进位标志。RCL 循环左移 DX 并把 CF 位插入到最右边已腾空的位。对于乘以 4, 用同样的 SHL-RCL 对跟着上面的 SHL-RCL 对即可。

对于除法, 仍然考虑在 DX: AX 中的 32 位值。该值除以 2 的指令是:

SAR DX, 1 ; 使用右移将
RCR AX, 1 ; DX: AX 对除以 2

SAR 操作把 DX 中的所有位向右移，并且最右边的位移入进位标志。RCR 循环右移 AX 并把 CF 位插入到最左边已腾空的位。对于除以 4，用同样的对跟着 SAR-RCR 对。

7.13 组织一个程序

以下是在编写汇编程序时推荐的步骤：

1. 对于程序要解决的问题有清晰的思路。

2. 概括地叙述你的思路并计划总的逻辑。例如，如果问题要完成多字节传送操作，那么开始要定义所要传送的字段。然后规划所用指令的策略：用于初始化的例行程序，用于条件转移的例行程序，以及使用循环的例行程序。下面说明的是主逻辑，它是许多程序员用于规划一个程序的伪代码：

初始化段寄存器。

调用转移例行程序。

调用循环例行程序。

结束处理。

转移例行程序可以规划如下：

初始化用于计数的寄存器，以及名为 JMP1：的地址。

传送名字的一个字符。

为名字的下一个字符而增量。

减量计数：如为非零，则 JMP1；

如为零，则返回。

循环例行程序可按类似的方法处理。

3. 把程序组织成为一些逻辑单元，它们是一个接一个的相关的例行程序。过程大约在 25 行左右(屏幕的大小)，这样的过程比更长的过程容易调试一些。

4. 用其他程序作为指导，试图记住所有技术资料并且想编写一个“别出心裁”的程序，通常会导致更多程序缺陷的产生。

5. 使用注释清楚地说明正在执行的算术和比较操作在做什么，以及很少用的指令是做什么的(后者的例子是 LOOPNE：当不相等时循环或是直到不相等时循环)。通常的做法是，为每个过程提供一个标题注释来说明它的用途，输入和输出，以及改变的寄存器。

6. 为了便于键入程序，可以用已保存的框架程序，你可以把它复制到一个新命名的文件中。

7. 开发一些测试模块，尽管它们当中许多是简单的空过程。这种实践有助于故障的早期定位与识别。

8. 使用数字数据进行最终测试，这些数据应该是有效值，包括零和负的情况。然后再用一些非常的数据试一下，看一看该程序如何处理它。

在本课本中，余下的程序会大量地使用 JMP，LOOP，条件转移，CALL，以及调用过程。有了前面已叙述过的汇编语言基础知识，你现在处于更为先进和实际的程序设计位置上。

7.14 要 点

- 短地址是通过偏移值到达的，被限制在-128 到 127 个字节的距离内。近地址通过偏移值到达并且距离被限制在同一段内的-32768 到 32767 个字节。在另一个段中的远地址，是通过段地址和偏移值到达的。
- 指令标号(比如 L10:)需要冒号指明它是一个近标号。
- 条件转移和 LOOP 指令的标号必须是短的。操作数产生 1 个字节的目標码：01H 到 7FH, 所覆盖的范围从十进制的+1 到+127, 而 FFH 到 80H 所覆盖的范围从-1 到-128。
- 当使用 LOOP 时，用一个正的非零值初始化 CX，因为 LOOP 使 CX 减 1，然后检查它是否为零。
- 当一条指令设置标志时，标志保存这一设置直到另一条指令改变它为止。
- 选择合适的条件转移指令，取决于操作处理的是无符号还是带符号数据。
- 使用 CALL 调用一个过程，在过程结束时应包含为返回而存在的 RET/RETN。被调用的过程可以调用其他过程，并且如果遵循习惯，RET 可使堆栈中的正确地址出栈。
- 使用左移加倍一个值，用右移减半一个值。一定要为无符号和带符号数据选择合适的移位和循环指令。

7.15 习 题

7-1. 解释这些地址类型的含义：(a)短的，(b)近的，(c)远的。

7-2. (a)近的 JMP，LOOP 和条件转移的最大字节数是多少？(b)产生这一限制的机器码操作数的特点是什么？

7-3. JMP 指令从偏移单元 02D4H 开始。根据以下 JMP 操作数的目标码：(a)2AH，(b)6EH，(c)B8H，确定传送的偏移地址。

7-4. 说明以下每个 LOOP 操作循环多少次：

(a) MOV CX, 1	(b) L5: MOV CX, 10
L5:	...
...	LOOP L5
LOOP L5	
(c) MOV CX, 0	(d) MOV CX, 10
L5:	L5:
...	INC CX
LOOP L5	LOOP L5

7-5. 编写计算 Fibonacci 序列：1, 1, 2, 3, 5, 8, 13, ... (除了序列中最先 2 个数之外，其余每个数都是前面 2 个数之和)的程序。使用 LOOP 并设置 12 次迭代的限制。汇编，连接，并使用 DEBUG 跟踪整个例行程序。

7-6. 编写程序，把在 BYTE_TBL 中定义每个值相加并把和存放在 BYTE-TOTAL 中。

(提示: 使用间接寻址, 一条 ADD, INC BX, 以及 LOOP。)

```
BYTE_TBL DB 5, 6, 4, 9, 7
BYTE-TOTAL DB 0
```

7-7. 编写程序, 把 CHAR_STRING 中所包含的大写字母转换成小写字母。逐次把每个字符取到寄存器中, 加上 20H, 并把它重新存放到该串中。使用间接寻址和 LOOP 指令。(提示: 使用间接寻址并为逐次取字符而使 BX 增 1。)

```
CHAR_STRING DB 'ABCDEFGHIJ'
```

7-8. 假定 CX 和 DX 含有无符号数据, AX 和 BX 含有带符号数据。为以下问题确定 CMP (这是必要的) 和条件转移指令: (a) AX 等于或小于 BX 吗? (b) CX 等于或小于 DX 吗? (c) AX 大于 BX 吗? (d) CX 大于 DX 吗? (e) DX 含有零吗? (f) 有溢出吗?

7-9. 在以下情况下, 受影响的标志是什么? 它们包含的内容是什么?

(a) 按单步方式处理。 (b) 串数据的传送是从右到左。 (c) 结果是负的。 (d) 结果是零。 (e) 发生了溢出。

7-10. 编写并测试一个程序, 该程序包含下面一组指令。确定目的寄存器中的值和在标志寄存器 OF, ZF, SF 及 CF 中的结果:

```
(a) MOV AL, FFH      (b) MOV BL, 24H
    ADD AL, 1          SUB BL, BL
(c) MOV CL, 10101010B (d) MOV DL, 01111100B
    ADD CL, 01010101B  ADD DL, 01110011B
```

7-11. 参考图 7-3 并说明如果过程 B10 没有包含 RET, 程序的执行结果如何。

7-12. 说明所定义的 PROC 操作数用 NEAR 和用 FAR 之间的区别。

7-13. 指出使正在执行的程序进入过程的 3 种方法。

7-14. 汇编并连接图 7-3 中的程序。先按 Small 存储模型, 再按 Medium 存储模型。比较所产生的目标码和连接映像的区别。

7-15. 在一个程序中, D10 调用 E10, E10 调用 F10, F10 调用 G10。作为这些调用的结果, 现在堆栈中包含多少地址?

7-16. 假定 DL 内容是 01111001, 名为 BOOT_AMT 的项目内容是 11100011。为以下不相关操作确定 DL 中的结果: (a) AND DL, BOOT_AMT, (b) OR DL, BOOT_AMT,

(c) XOR DL, BOOT_AMT, (d) AND DL, 00000000B, (e) XOR DL, 11111111B。

7-17. 按以下要求修改图 7-4 的程序: 定义 CONAME 的内容为所有都是小写字母, 并且编写指令把所有小写字母转换成大写字母。

7-18. 假定 DX 的内容是二进制的 10111001 10111001。在以下不相关的指令执行之后, 确定 DX 的二进制内容: (a) SHL DL, 1, (b) SHL DX, 2, (c) SHR DX, 1, (d) SAR DX, 2, (e) SAL DH, 3, (f) ROR DX, 3, (g) ROR DL, 3。

7-19. 使用传送(mov)、移位(shift)和加法(add)指令, 用 24H 初始化 DX 并把它乘以 10。

7-20. 在标题为“循环移位”一节的末尾有个例子是 DX: AX 乘以 2。修改该子程序为 (a) 乘以 4, (b) 除以 4, (c) 把 DX:AX:BX 中的 48 位值乘以 2。

7-21. 把 DX: CX 的内容传送到 EBX 并使用移位实现 EBX 乘以 4。

第三部分

视频与键盘操作

目的：介绍屏幕显示信息和接收键盘输入的中断请求。

8.1 引言

本章之前的程序都是在数据区定义数据项，或者在指令操作数字段给定立即数。然而，大多数程序都要求从外部设备输入源数据，例如键盘、磁盘、鼠标或调制解调器，并且提供有效的格式输出到屏幕、打印机或磁盘上。本章包含了屏幕显示信息和接收键盘输入的基本中断请求。

INT(中断)指令能处理大部分的输入输出请求。本章包含的两个中断是屏幕处理功能 INT 10H 以及屏幕显示输出和接收键盘输入的 INT 21H 功能。这些功能要求一个特定的动作，在 AH 寄存器中插入一个功能值，以识别要运行的中断服务程序的类型。

低级 BIOS 操作，如 INT 10H 直接转移控制到 BIOS。然而，为了便于一些更复杂的操作，INT 21H 提供了先把控制转移给操作系统的中断服务。例如，键盘输入可包括一定数量的输入字符以及对最大数量的检测。INT 21H 操作完成很多这种附加的高级处理，然后自动地把控制转移给 BIOS，由 BIOS 来处理操作的低级部分。

作为惯例，这本书所提到的 0DH 即是键盘输入以及屏幕和打印机的回车符。

本章介绍的 INT 10H 和 INT 21H 操作有：

INT 10H 功能

02H 设置光标
06H 屏幕滚动

INT 21H 功能

02H 显示字符
09H 显示串
0AH 键盘输入
3FH 键盘输入
40H 显示串

第 9 章和第 10 章将包含更高级的屏幕和键盘处理特性。

8.2 屏幕特征

一个典型的视频显示器有 25 行(从 0 到 24)和 80 列(从 0 到 79)。行和列为设置光标提供了一个可寻址的网格。以下是光标位置的一些例子：

屏幕位置	十进制		十六进制	
	行	列	行	列
左上角	00	00	00H	00H
右上角	00	79	00H	4FH
屏幕中心	12	39/40	0CH	27H/28H
左下角	24	00	18H	00H
右下角	24	79	18H	4FH

系统为视频显示区在内存里提供了一块空间或缓冲区。它在 BIOS 中的位置是不同的，这取决于系统当前的操作方式，比如，文本或图形方式以及彩色或单色方式。在文本方式下，视频显示区需要 4K 字节内存，其中 2K 用于存储字符，2K 用于存储每个字符的属性，如反相显示、闪烁、高亮度和下划线等。

视频显示区还提供屏幕“页”，编号从 0 到 7。这些页及其属性会在第 10 章中详细介绍，在本章我们假定显示页为默认的 0 页。

处理屏幕显示的中断功能直接将数据传输到视频显示区。从技术上讲，程序可以直接传输数据到视频显示区。但是，存储器地址对不同的显示模式是不同的，而且也不能保证地址总是不变，所以直接往显示区写数据虽然比较快，但可能会有些危险。通常的做法是使用适当的 INT 10H 和 INT 21H 中断操作，这些中断是知道视频显示区的位置的。

8.3 设置光标

设置光标是一个基本的中断请求，因为光标的位置决定了下一个字符将在哪里显示或输入。屏幕处理的 BIOS 操作取决于 INT 10H 以及 AH 中的功能码。例如，INT 10H 的 02H 功能是告诉 BIOS 设置光标，并将要求的页号装入 BH(页号通常为 0)，行号装入 DH，列号装入 DL。其他寄存器的内容不重要。

下面的例子将光标设置在第 8 行，第 15 列：

```
MOV AH, 02H      ; 请求设置光标
MOV BH, 00       ; 页号 0
MOV DH, 08       ; 第 8 行
MOV DL, 15       ; 第 15 列
INT 10H          ; 调用中断服务
```

也可以用一条带十六进制立即数的 MOV 指令在 DX 中设置行和列：

```
MOV DX, 080FH    ; 第 8 行，15 列
```

8.4 清 屏

INT 10H 的 06H 功能处理屏幕清除或滚动。清除全屏或部分屏幕显示，可在全屏范围内的任意位置开始和任意位置结束。设置以下寄存器：

AH=06H 功能号
 AL=滚动的行数, 或 00H 全屏为空白
 BH=属性值(颜色、闪烁等)
 CX=起始行:列
 DX=结束行:列

CX 和 DX 共同定义了滚动的屏幕区(或窗口), AL 指明上卷的行数。如果要清除整个屏幕, 就指定 CX 中的起始行:列为 00: 00H, DX 中的结束行:列为 18:4FH。下例中的属性 71H 设置整个屏幕为白色背景(属性为 7)和蓝色前景(属性为 1):

```
MOV AX, 0600H    ; AH=06(滚动), AL=00(全屏空白)
MOV BH, 71H      ; 白色背景(7), 蓝色前景(1)
MOV CX, 0000H    ; 左上角行:列
MOV DX, 184FH    ; 右下角行:列
INT 10H          ; 调用中断服务
```

例如, 要在一个从 05 行、00 列到 12 行、79 列的屏幕窗口中滚动, 就在 CX 中装入 0500H, 在 DX 中装入 0C4FH。

注意不要错误地设置右下角屏幕位置大于 184FH。下一章将会更详细地讲述滚屏。

程序经常要显示信息, 以要求用户输入数据或做一些动作。我们首先练习一些编写小程序的简单方法, 随后再练习一些包含文件处理的方法。

8.5 屏幕显示的 INT 21H 功能 09H

简单的 INT 21H 的 09H 显示功能对初学者来说是很方便的, 它要求在数据区定义一个显示串, 紧接着用一个美元符(\$或 24H)作为定界符来结束显示操作。这种方法的缺点是你无法用这个功能在屏幕上显示\$字符。下面举例说明:

```
CUST_MSG DB 'Customer name?', '$' ; 显示串
```

你可以像上面的例子一样紧接着显示串加上美元符, 也可以在串内加美元符, 如 'Customer name?\$', 或在下一行加美元符, 如 DB '\$'。在 AH 中设置 09H 功能, 用 LEA 指令在 DX 中装入显示串地址, 然后发出 INT 21H 指令:

```
MOV AH, 09H      ; 请求显示
LEA DX, CUST_MSG ; 装入提示符地址
INT 21H          ; 调用中断服务
```

INT 操作从左到右显示字符, 美元定界符(\$)作为数据结束标志。这个操作不改变寄存器的内容。如果一个显示串超过了屏幕最右边的列, 可自动在下一行继续显示, 并在必要的时候滚动屏幕。如果在串尾省略了\$符, 该操作将不断显示连续存储空间中的字符直到遇到一个\$——如果存在一个\$的话。

显示 ASCII 字符

256 个 ASCII 字符中的大多数是通过符号来表示的, 这些符号能在视频屏幕上显示, 而

对于某些值, 比如 00H 和 FFH 没有可显示的符号, 所以显现为空白, 虽然实际上的 ASCII 空白符是 20H。

图 8-1 的程序显示全部 ASCII 字符。程序 A10MAIN 调用 3 个过程:

```

TITLE      A08DISAS (COM)  Display ASCII character set
.MODEL     SMALL
.CODE
BEGIN:     ORG      100H
ASCHAR     JMP      SHORT A10MAIN
           DB       00, '$'
           ;        Main procedure:
           ;        -----
           .286
A10MAIN    PROC     NEAR
           CALL     B10SCREEN      ; 清屏
           CALL     C10CURSOR     ; 设置光标
           CALL     D10DISPLY     ; 显示字符
           MOV      AX, 4C00H     ; 程序结束
           INT      21H
A10MAIN    ENDP

           ;        清屏并设置光标:
           ;        -----
B10SCREEN  PROC     NEAR
           PUSH     AX            ; 保存通用寄存器
           MOV      AX, 0600H    ; 清除全屏
           MOV      BH, 07       ; 属性: 黑底白字
           MOV      CX, 0000     ; 左上角位置
           MOV      DX, 184FH    ; 右下角位置
           INT      10H          ; 调用 BIOS
           POPA     ; 恢复通用寄存器
           RET         ; 返回调用程序
B10SCREEN  ENDP

           ;        设置光标为 08,00:
           ;        -----
C10CURSOR PROC     NEAR
           PUSH     AX            ; 保存通用寄存器
           MOV      AH, 02H      ; 请求设置光标
           MOV      BH, 00       ; 页号 0
           MOV      DX, 0800H    ; 8 行, 0 列
           INT      10H          ; 调用 BIOS
           POPA     ; 恢复通用寄存器
           RET         ; 返回调用程序
C10CURSOR  ENDP

           ;        显示 ASCII 字符 00H - FFH,
           ;        跳过 08H - 0DH 之间的字符
           ;        -----
D10DISPLY  PROC     NEAR
           PUSH     AX            ; 保存通用寄存器
           MOV      CX, 256      ; 初始化为 256 次循环
           LEA      DX, ASCHAR    ; ASCHAR 的起始地址
D20:       CMP      ASCHAR, 08H   ; 低于 08H 的字符?
           JB       D30          ; 是, 则接收
           CMP      ASCHAR, 0DH   ; 低于或等于 0DH?
           JBE      D40          ; 是, 则跳过
D30:       MOV      AH, 09H      ; 显示 ASCII 字符
           INT      21H
D40:       INC      ASCHAR        ; 加1为下一个字符
           LOOP     D20          ; CX减1, 不为0则循环
           POPA     ; 恢复通用寄存器
           RET         ; 返回到调用程序
D10DISPLY  ENDP
END        BEGIN

```

图 8-1 显示 ASCII 字符集

- B10SCREEN 调用 INT 10H 的功能 06H 清屏。
- C10CURSOR 调用 INT 10H 的功能 02H 把光标初始为 08:00H。

- D10DISPLY 调用 INT 21H 的功能 09H 显示 ASCHAR 的内容, 显示内容先初始化为 00H, 然后连续增 1 直到 FFH 显示每个 ASCII 字符。

显示的第一行以一个空白(00H)、两个“笑脸”(01H 和 02H)和一个红心(03H)、红方块(04H)、梅花(05H)、黑桃(06H)开始, ASCII 07H 使扬声器发声。程序跳过从 08H 到 0DH 的所有字符(08H 将产生一个退格, 09H 产生一个 TAB 键, 0AH 产生的换行以及 0DH 产生的“回车”使显示从下一行开始)。当然, 在功能 09H 中 \$ 符号(24H)是根本不显示出来的(在第 9 章, 你可以调用 BIOS 服务以特定的符号来显示这些特殊字符)。乐符为 0EH, 从 7FH 到 FFH 是扩展 ASCII 字符。

注意, 显示退格符、Tab 符、换行和回车符是执行这些操作的常用方法。

建议: 复制先前的程序, 然后编译、连接并转换成一个 .COM 文件执行。

8.6 键盘输入的 INT 21H 功能 0AH

从键盘接收数据的 INT 21H 的 0AH 功能特别强。键入字符的输入区需要一个包含指定区域的参数表以便进行 INT 操作 (相当于高级语言里的记录或结构)。首先, 该操作需要知道输入字符的最大数量。其目的是防止用户从键盘输入过多的字符, 如果超过最大数, 这个操作使扬声器发出声音并且不再接收字符。第二步, 把实际输入的字节数传送到参数表。参数表包含这些元素:

1. 第一个语句以 LABEL BYTE 的格式提供参数表的名字。LABEL 是带有 BYTE 类型属性的指令, 它简单地使参数表按字节边界来定位, 因为这是最常用的定位, 汇编程序不必预置它的地址计数器。LABEL 用来给参数表分配一个名字。

2. 参数表的第一个字节限制输入字符的最大数。这个数最小是 0, 最大是 FFH 即 255, 因为这是一个 1 字节的字段。可以对最大字符数作出决定, 最大字符数是根据用户输入的数据类型而定的。

3. 此操作的第二个字节是以二进制形式存储实际输入的字符数。

4. 从第三个字节开始的区域, 从左到右存放键入的字符。

下面的例子是为一个键盘输入区定义的参数表:

```

PARAM_LIST LABEL BYTE           ; 参数表开始
MAX_LEN    DB 20                 ; 最大输入字符个数
ACT_LEN    DB ?                  ; 实际输入字符数
KB_DATA    DB 20 DUP(' ')       ; 从键盘输入的字符
  
```

在参数表中, LABEL 指令告诉汇编程序定位一个字节边界并且赋地址名 PARAM_LIST。因为 LABEL 不占用空间, PARAM_LIST 和 MAX_LEN 指向同一个内存地址。MAX_LEN 定义了键盘字符的最大数(20), ACT_LEN 为操作填入实际输入的字符数提供了空间, KB_DATA 为字符保留了 20 个字节空间。对这些字段你可以使用任何一个有效的名字。

为了请求键盘输入, 要在 AH 中设置功能 0AH, 在 DX 中装入参数表地址(在上例中是 PARAM_LIST), 并发出 INT 21H 指令:

```

MOV AH, 0AH           ; 请求键盘输入
  
```

```
LEA    DX, PARA_LIST    ; 装入参数表地址
INT    21H               ; 调用中断服务
```

INT 操作等待用户输入字符并检测输入字符数是否超过最大数 20。此操作回送每个字符到屏幕上光标所在的位置,并右移光标。用户按<Enter>键作为键盘输入结束的信号。这个操作也传送回车符(0DH)到输入区 KB_DATA,但不把它记入实际长度。若键入一个名字,如 Wilson+<Enter>,参数表显示如下:

ASCII:	20	6	w	i	l	s	o	n	#				...
Hex:	14	06	57	69	6C	73	6F	6E	0D	20	20	20	...

该操作把输入名字的长度 06H 送入参数表的第二个字节,即例子中命名的 ACT_LEN。回车字符(0DH)在 KB_DATA+6 中(这里 # 号就表示这个字符,因为 0DH 没有可打印的符号)。给出的最大字符长度 20 包括了 0DH,用户最多只能敲入 19 个字符。

此操作能接收退格并作出退格动作,但是不把它加入字符数。除了退格,此操作不接收多于最大字符数的字符。在前面的例子中,如果一名用户键入 20 个字符却没有按回车,该操作就会使扬声器发出“嘀”的声音,在此刻它只接收回车符。

虽然这个操作对输入数据是很有用的,但它无法使用扩展的功能键,如 F1、Home、PgUp 和 Arrows。如果能够预料用户会按动它们中的一个,请使用 INT 16H 或 INT 21H 的功能 01H,这将在第 10 章中详细讲述。

8.6.1 程序:接收并显示名字

图 8-2 的程序要求用户键入一个名字,然后在屏幕中央显示这个名字,并使扬声器发声。如果用户敲入 Dana Wilson,程序执行如下:

1. 长度 11 除 2: $11/2=5$, 忽略余数。
2. 从 40 中减去这个值: $40-5=35$ 。

在 C10CENTER 中,SHR 指令把长度值 11 向右移一位,相当于长度除以 2: 字节 00001011 成为 00000101(即 5)。NEG 指令把符号变反,使 +5 变成 -5。ADD 加 40,在 DL 中得到起始位置的列号 35。因为光标设置在 12 行、35 列,名字就显示在第 12 行、第 35 列开始的屏幕位置上。

注意 C10CENTER 在紧接着名字的输入区插入响铃符(07H):

```
MOVZX  BX, ACTULEN      ; 用 07H 替换 0DH
MOV     KBNAM[BX], 07H
```

MOVZX 设置 BX 为已键入的字符数。在 MOV 指令中, [BX] 作为一个变址寄存器用于扩大寻址范围。MOV 指令把 BX 中的长度与 KBNAM 的地址组合,并把 07H 传送到计算出来的地址中去。因为长度为 11,指令在名字后面的 KBNAM+11 单元插入 07H(替代回车符)。C10CENTER 中的指令

```
MOV     KBNAM[BX+1], '$' ; 设置显示定界符
```

在 07H 之后插入一个 '\$',这样 INT 21H 的功能 09H 就能显示名字并使扬声器发声。

过程 C10CENTER、D10DISPLY 和 Q20CURSOR 通过注释来说明寄存器的用途。实际上

这些过程应当 PUSH 和 POP 所用寄存器的内容。

8.6.2 附加的编程练习

这一节介绍几个很有用的技术。

1. 用回车键作为回答信号。图 8-2 的程序连续地接收和显示名字，直到用户按动了<回车>才作为对一个提示的回答。INT 21H 的功能 09H 接收按键，并在参数表中插入长度 00H，如下所示：

参数表(16 进制):

14	00	0D	---
----	----	----	-----

```

TITLE      A08CTRM (EXE) 从键盘接收名字
;          在屏幕中央显示名字，并响铃

.MODEL SMALL
.STACK 64

.DATA
PARLIST LABEL BYTE ; 名字的参数表,
MAXLEN DB 20 ; 名字的最大长度
ACTULEN DB ? ; 输入的字符数
KBNAME DB 21 DUP(' ') ; 存入名字
PROMPT DB 'Name? ', '$'
;-----
.CODE
.386 ; 为 MOVZX 指令指定处理器
A10MAIN PROC FAR
MOV AX,@data ; 初始化段寄存器
MOV DS,AX
MOV ES,AX
CALL Q10CLEAR ; 清屏

A20: MOV DX,0000 ; 设置光标为 00,00
CALL Q20CURSOR
CALL B10INPUT ; 用于输入名字
CALL Q10CLEAR ; 清屏
CMP ACTULEN,00 ; 输入名字?
JE A30 ; 否, 则退出
CALL C10CENTER ; 设置响铃和 '$'
CALL D10DISPLY ; 并在中央显示名字
JMP A20 ; 重复

A30: MOV AX,4C00H ; 处理结束
INT 21H

A10MAIN ENDP
;
;----- 显示提示符并接收输入的名字:
;-----
B10INPUT PROC NEAR
PUSH AX ; 保存要使用的
PUSH DX ; 寄存器
MOV AH,09H ; 请求显示用户提示符
LEA DX,PROMPT
INT 21H
MOV AH,0AH ; 请求键盘输入
LEA DX,PARLIST
INT 21H
POP DX ; 恢复寄存器
POP AX
RET
B10INPUT ENDP
;
;----- 设置响铃和 '$' 定界符
;----- 在屏幕中央设置光标:
;-----
C10CENTER PROC NEAR ; 使用 BX 和 DX
MOVZX BX,ACTULEN ; 用 07H 替换 0DH
MOV KBNAME[BX],07
MOV KBNAME[BX+1],'$' ; 设置显示定界符
MOV DL,ACTULEN ; 定位中心位置的列:
SHR DL,1 ; 长度除 2,

```

图 8-2 接收并显示名字

```

        NEG     DL                      ; 符号要求
        ADD     DL, 40                  ; 加 40
        MOV     DH, 12                 ; 中心位置的引号
        CALL    Q20CURSOR              ; 设置光标
        RET
C10CENTER ENDP
;
; ----- 显示中心位置的名字: -----
D10DISPLY PROC NEAR                   ; 使用 AH 和 DX
        MOV     AH, 09H
        LEA     DX, KBNNAME           ; 显示名字
        INT     21H
        RET
D10DISPLY ENDP
;
; ----- 清屏并设置属性: -----
Q10CLEAR  PROC NEAR
        PUSH    AX                    ; 保存通用寄存器
        MOV     AX, 0600H             ; 请求卷屏
        MOV     BH, 30                ; 彩色属性
        MOV     CX, 0000             ; 从 00, 00
        MOV     DX, 184FH            ; 到 24, 79
        POPA
        INT     10H                  ; 恢复通用寄存器
        RET
Q10CLEAR  ENDP
;
; ----- 设置光标的行列: -----
Q20CURSOR PROC NEAR                   ; DX 已输入数据
        MOV     AH, 02H               ; 使用 AH 和 BH
        MOV     BH, 00                ; 请求设置光标
        INT     10H                  ; 页号 0
        RET
Q20CURSOR ENDP
END      A10MAIN

```

图 8-2 续

如果是串的长度为 0，程序就确定输入结束，程序 A10MAIN 通过指令 CMP ACTLEN, 00 表明了这个判断过程。

2. 清除回车符。可能为不同的目的输入字符，例如打一份报告，存入一个表格，或写入磁盘。对这些用途，可能必须用空格(20H)来取代回车符，无论它在 KBNNAME 的什么位置。含有输入数据实际长度的字段 ACTLEN 提供了回车符的相对位置。例如，若 ACTLEN 含有 11，那么回车符就在 KBNNAME+11 的位置。可以传送这个长度给 BX，以标定 KBNNAME 中地址，如下所示：

```

MOVZX    BX, ACTLEN      ; 设置 BX 为 00 0B(11)
MOV      KBNNAME[BX], 20H ; 清除回车符

```

MOVZX 指令设置 BX 长度为 11。MOV 指令把一个空格符(20H)传送给第一个操作数指定的地址：KBNNAME 的地址加上 BX 的内容，即 KBNNAME+11。

3. 清除键盘输入区。键入的每个字符都会取代输入区中先前的内容，并且保留在那里直到其他的字符取代了它们。我们来看下面的连续输入：

输入

参数表(十六进制)

1. Monroe	14	06	4D	6F	6E	72	6F	65	0D	20	20	...	20
2. Franklin	14	08	46	72	61	6E	5B	6C	69	6E	0D	...	20
3. Adams	14	05	41	64	61	6D	73	0D	69	6E	0D	...	20

第一个名字 **Monroe** 只需要 6 个字节。第二个名字 **Franklin** 全部替代了较短的名字 **Monroe**。但是因为第三个名字 **Adams** 比 **Franklin** 短，它只替代了 **Frank**，回车符代替了 **l**，剩余的两个字母(“in”)仍然保留在 **Adams** 之后。可以预先清除 **KBNAME** 再提示输入名字，如下所示：

```

MOV     CX, 20                ; 初始化循环次数 20
MOV     SI, 0000              ; 名字的起始地址
L10:
MOV     KBNAME[SI], 20H        ; 传送一个空白给名字区
INC     SI                    ; 为指向下一个字符增 1
LOOP    L10                   ; 重复 20 次

```

可以用 **DI** 或 **BX** 代替 **SI**。如果程序传送两个空白符组成的一个字，则只需要 10 次循环。然而因为 **KBNAME** 已经定义为 **DB**(字节)，若使用 **WORD** 和 **PTR(pointer)**操作数，则不必考虑它的长度，如下所示：

```

MOV     CX, 10                ; 初始化循环次数 10
LEA     SI, KBNAME            ; 初始化名字区的起始地址
L10:
MOV     WORD PTR[SI], 2020H    ; 传送 2 个空白到名字区
ADD     SI, 2                  ; 增 2 指向下一个名字区位置
LOOP    L10                   ; 重复 10 次

```

在 **L10** 的 **MOV** 指令可解释为：“传送一个空白字给 **SI** 指向的存储器单元”。这个例子使用 **LEA** 指令来初始化要清除的 **KBNAME**，并对 **L10** 的 **MOV** 指令使用了不同的方法，不能编写如下的指令：

```
MOV WORD PTR[KBNAME], 2020H ; 第一个操作数是非法的
```

清除输入区解决了短名字之后仍保留先前数据的问题。为了更快地处理，也可以只清除紧接着输入名字右面的区域。

8.6.3 用于屏幕显示的控制字符

使显示更加有效的一种方法是利用回车、换行和 **Tab** 控制符，可以用它们的 **ASCII** 码或十六进制值来编码，如下所示：

控制字符	ASCII 码	16 进制数	对光标的影响
回车	13	0DH	重新回到屏幕左边的位置
换行	10	0AH	前进到下一行
Tab	09	09H	前进到下一个 Tab 位置停止

在显示输出或接收键盘输入的任何时候，都可使用控制字符来处理光标。下例在显示一个字符串 **REPTITLE** 的内容后，紧接其后的回车(13)和换行(10)设置光标到下一行。利用 **EQU** 命令重新定义控制符能增强程序的可读性：

```

CR      EQU 13                ; 或 EQU 0DH
LF      EQU 10                ; 或 EQU 0AH

```

```

TAB      EQU 09      ; 或 EQU 09H
REPTITLE DB TAB,     'Annual Rainfall Statistics', CR, LF, '$'
...
MOV AH, 09H      ; 请求显示功能
LEA DX, REPTITLE ; 装入字符串地址
INT 21H          ; 调用中断服务

```

8.7 屏幕显示的 INT 21H 功能 02H

INT 21H 的功能 02H 用于显示单字符。在 DL 中装入在当前光标位置上要显示的字符，然后请求 INT 21H。Tab、回车和换行的动作照常，该操作自动前移光标。指令如下：

```

MOV AH, 02H      ; 请求显示字符
MOV DL, char      ; 要显示的字符
INT 21H          ; 调用中断服务

```

下面的例子说明了如何利用这个中断服务来显示一串字符。要显示的字符串定义在 CO_TITLE。该例把 CO_TITLE 的地址装入 DI，其长度装入 CX。为指向每一个连续的字符，循环中包括了 DI 增 1(通过 INC 指令)。为了控制要显示的字符数，还包括了 CX 减 1(通过 LOOP 指令)。下面是指令序列：

```

CO_TITLE DB 'Intertech Corp.', 13, 10
...
MOV AH, 02H      ; 请求显示字符
MOV CX, 17       ; 字符串长度
LEA DI, COTITLE   ; 字符串地址
L10: MOV DL, [DI]  ; 要显示的字符
      INT 21H      ; 调用中断服务
      INC DI       ; 增量指向下一个字符
      LOOP L10     ; 重复 17 次
...              ; 结束

```

8.8 文件代号

本节主要讲解有关屏幕和键盘操作的文件代号的用法。文件代号是对应于特定设备的一个简单的数字。以下的标准文件代号是事先设置好的，不需要再定义：

文件代号	设备
00	输入设备，通常是键盘(CON)，可以重定向
01	输出设备，通常是显示器(CON)，可以重定向
02	错误输出设备，一般为显示器(CON)，不能重定向
03	辅助设备(AUX)
04	打印设备(LPT1 或 PRN)

如上所示，键盘的文件代号是 00，屏幕显示的文件代号是 01。磁盘设备的文件代号(包含在 17 章)必须由程序来设置。也可以利用这些中断服务把输入输出重定向到其他设备，重定向的特性我们这里没有涉及到。

8.9 屏幕显示的 INT 21H 功能 40H

INT 21H 的功能 40H 利用文件代号来处理显示操作。为了请求这个中断服务，要设置以下寄存器：

AH=功能 40H	CX=要显示的字符数
BX=文件代号 01	DX=显示区的地址

若 INT 操作成功，则把写入的字节数传送给 AX，并清除进位标志(可以测试出来)。

若 INT 操作不成功，则置进位标志为 1，并在 AX 中返回一个错误代码：05H=拒绝存取(无效的或未连接的设备)，或 06H=无效文件代号。由于 AX 中既可能包含字节的长度又可能包含错误代码，所以确定出错条件的唯一办法就是测试进位标志。通常情况下不会发生显示出错：

JC error_routine ; 测试显示错误

该操作对控制字符 07H(嘀嘀声)、08H(退格)、0AH(换行)和 0DH(回车)产生的动作就像 INT 21H 的功能 09H 一样。下面的指令说明 40H 的功能：

```
PROMPT      DB      'Part number?', 0DH, 0AH      ; 显示区
...
MOV AH, 40H      ; 请求显示中断
MOV BX, 01       ; 显示器的文件代号
MOV CX, 14       ; 14 个字符，包括 0DH+0AH
LEA DX, PROMPT   ; 显示区
INT 21H          ; 调用中断服务
```

练习：使用功能 40H 在屏幕上显示

让我们利用 DEBUG 来检验使用文件代号显示名字的内部作用。装载 DEBUG，当出现提示符时，敲入 A 100，并在偏移地址 100H(记住 DEBUG 认定输入的数字都是十六进制格式的)开始键入下面的汇编语句(不要键入最左边的数字)：

```
100      MOV AH, 40
102      MOV BX, 01
105      MOV CX, xx(以 16 进制插入名字的长度)
108      MOV DX, 10FH
10B      INT 21H
10D      JMP 100
10F      DB  'x----x'(插入你的名字)
```

指令设置 AH 以请求一次显示中断，并在 DX 中设置偏移地址 10FH——DB 定义的单元中含有你的名字。

键入指令后, 按<Enter>。如果要反汇编程序, 使用 U 命令(U 100, 10F), 要跟踪执行, 按下 R, 然后反复使用 T 命令。当运行到 INT 21, 用 P(Proceed)命令执行中断, 直到 JMP 指令, 此时在屏幕上应该显示出你的名字了。

8.10 键盘输入的 INT 21H 功能 3FH

INT 21H 的功能 3FH 利用文件代号来请求键盘输入, 虽然这是有些笨拙的操作。装载下面的寄存器:

AX=功能号 3FH	CX=要接收的最大字符数
BX=文件代号 00	DX=输入字符区的地址

若 INT 操作成功, 则清除进位标志(这可以测试出来), 并设置 AX 为输入的字符数。

发生不成功的 INT 操作可能是因为使用了无效的文件代号; 此时该操作置进位标志为 1, 并在 AX 中插入一个错误代码: 05H=拒绝存取(对一个无效设备或未连接的设备), 或 06H=无效文件代号。因为 AX 中既可能是字符的长度又可能是错误代码, 所以确定出错条件的唯一办法就是测试进位标志, 虽然键盘出错的机会很少。

类似 INT 21H 的功能 0AH, 功能 3FH 也能对退格起作用, 但是对扩展功能键如 F1、Home 和 PageUp 却不起作用, 因此极大地限制了其有效性。

以下的指令说明了功能 3FH 的用法:

KBINPUT	DB 20 DUP(' ')	: 输入区
...		
MOV AH, 3FH		: 请求键盘输入功能
MOV BX, 00		: 键盘的文件代号
MOV CX, 20		: 最大字符数 20
LEA DX, KBINPUT		: 输入区
INT 21H		: 调用中断服务

INT 操作等待你输入字符, 但不幸的是它不能检测字符数是否超过了 CX 中的最大数(本例中是 20)。按下回车键(0DH)表示输入结束。例如, 输入字符“Intertech Corp”, 则传送下列字符给 KBINPUT:

Intertech Corp	0DH	0AH
----------------	-----	-----

输入字符后面紧接着回车符(0DH)和换行符(0AH), 回车是你键入的, 而换行不是。由于这个特点, 你定义的最大字符数和输入区的长度应提供这两个额外字符的空间。如果键入的字符少于最大数, 在输入字符后面的存储器单元仍然保留先前的内容。

若 INT 操作成功, 则清除进位标志并将 AX 设置为传送的字符数。在前面的例子中, 这个数是 14+2(回车和换行符)。因此, 程序可以利用 AX 的值确定实际键入的字符数。虽然这个返回值对键入 YES 和 NO 时所起的作用不大, 但是对键入像姓名这样的不定长度字符时, 返回值是很有用的。

如果键入的字符数超过了 CX 中的最大数, 操作实际上接受了全部字符。再看下面一种情况, CX 中的最大字符数是 08, 而用户键入字符“PC Exchange”。该操作将前 8 个字符“PC

Excha”放入输入区，后面没有回车和换行，AX 设置为长度 8。现在我们看到，执行下一条 INT 操作没有直接从键盘接受名字，因为在缓冲区仍保留有先前的字符串。于是“nge”及紧跟着的回车和换行传送到输入区，并将 AX 置为 05。若操作都“正常”则清除进位标志：

```
第 1 个 INT:      PC Excha      AX=08
第 2 个 INT:      nge, 0DH, 0AH  AX=05
```

程序可以知道用户是否键入了“有效的”字符数：(a)AX 中的返回值小于 CX 中的数，或(b)AX 中的返回值等于 CX 中的数，并且输入区最后两个字符是 0DH 和 0AH。如果这两个条件都不符合，则必须发出另外的 INT 指令来接收其余的字符。这样做之后，你可能会感到在 CX 中指定最大长度是多么的重要。

练习：利用功能 3FH 键入数据

在这里的 DEBUG 练习中可以观察到使用 INT 21H 的功能 3FH 键入数据的效果。程序允许键入最多 12 个字符，包括一个回车符和一个换行符。装入 DEBUG，出现提示符时，键入 A 100，从 100H 单元开始输入下面的指令(不要输入左边的数字)：

```
100    MOV     AH, 3F
102    MOV     BX, 00
105    MOV     CX, 0C
108    MOV     DX, 10F
10B    INT     21
10D    JMP     100
10F    DB     20 20 20 20 20 20 20 20 20 20 20 20
```

输入完指令再按回车键。程序设置 AH 和 BX 以请求键盘输入，并在 CX 中插入最大字符长度。程序还在 DX 中设置偏移地址 10FH——这是 DB 的地址，输入的字符就从这里开始存放。

试发出 U 命令(U 100, 10E)反汇编程序。用 R 命令和多次的 T 命令跟踪 4 条 MOV 指令的执行。在地址 10BH，用 P 命令来执行中断，该中断操作等待你键入字符及紧接着的回车。检查 AX 和进位标志的内容，并用 D DS:10F 命令显示存储器中的输入字符。可以连续循环执行这个过程。

8.11 要 点

- 处理屏幕显示，传输数据到视频显示区的中断。
- INT 10H 指令是传输控制给 BIOS 的显示操作。常用的两个操作是功能 02H(设置光标)和 06H(滚动屏幕)。
- 当使用 INT 21H 的功能 09H 来显示时，紧接着显示区要定义一个定界符(\$)。丢失定界符会在屏幕上引起意想不到的结果。
- INT 21H 键盘输入功能 0AH 请求一个参数表。第一个字节存放最大字符数，第二个字节填入实际字符数。

- 文件代号是一个对应特定设备的数。对于文件代号从 00 到 04 这些数是预置的，但是其余的文件代号可以由程序来设置。
- 用 INT 21H 功能 40H 来显示，在 BX 中置文件代号 01。
- 用 INT 21H 功能 3FH 来接收键盘输入，在 BX 中置文件代号 00。该操作在输入区的键入字符之后插入回车和换行符，但是不检验字符是否超出了指定的最大数。

8.12 习 题

8-1. 在一个 80 列的屏幕上，下列位置的 16 进制地址是什么？(a)屏幕底部最右边的位置，(b)屏幕顶部最左边的位置。

8-2. 写指令把光标设置在第 12 行，第 24 列。

8-3. 写指令将从第 08 行，第 0 列开始的位置清屏，使用彩色属性 71H。

8-4. 定义数据项并用 INT 21H 的功能 09H 来显示一个信息“What is the date(mm/dd/yy)?”并在显示信息后发出响声。

8-5. 定义数据项，并用 INT 21H 的功能 0AH 按照 8-4 题的格式来接收键盘输入。

8-6. 某节的标题“Clearing the Input Area”定义为 KBNNAME，说明如何将这个键盘输入区清除为空(blank)。再把本例修改为只清除紧接着输入名字右边的字符。

8-7. 按下列要求修改图 8-2 的程序：(a)允许名字的最大字符数为 25；(b)取代 12 行，把光标设置在第 7 行；(c)取代清屏，只清除 0 到 7 行，操作中使用属性 16H；(d)在每一个子过程的开始入栈保存所用的寄存器，在退出之前把它们出栈。汇编、连接并测试程序。

8-8. 写出下列设备的文件代号：(a)打印机，(b)键盘输入，(c)常用屏幕显示。

8-9. 定义数据项，并使用 INT 21H 的功能 40H 来显示信息“What is the date(mm/dd/yy)?”紧接着信息的是回车，换行和“响”声。

8-10. 在一个程序中，INT 21H 的功能 3FH 已经设置了进位标志，并在 AX 中返回 06H，引起这个错误的原因可能是什么？

8-11. 定义数据项，并利用 INT 21H 功能 3FH，按照 8-9 题的格式接收键盘输入。

8-12. 修改 8-7 题，使用 INT 21H 的功能 3FH 和 40H 来输入和显示。汇编、连接并测试程序。

目的：介绍卷屏，反相显示，设置显示方式和属性以及使用图形等屏幕处理的高级特性。

9.1 引言

本章将在第 8 章的基础上进一步介绍有关视频操作的内容。第一节描述了视频系统的组成：监视器，视频显示区和视频控制器。第二节解释说明了一些视频特性，如显示方式、属性以及页等。最主要的一节是关于 BIOS 视频操作 INT 10H 的，其中描述了很多功能，包括设置显示方式、设置光标、滚动屏幕、显示字符和设置属性等。

最后一小节介绍了一些用于显示图形的功能，以及直接视频显示和设计菜单框。

本章包括了以下 BIOS INT 10H 的中断服务：

00H	设置显示方式	0BH	设置彩色调色板
01H	设置光标大小	0CH	写像素
02H	设置光标位置	0DH	读像素
03H	返回光标状态	0EH	打字方式写
05H	选择活动页	0FH	取当前显示方式
06H	上卷屏幕	10H	存取调色板寄存器
07H	下卷屏幕	11H	存取字符发生器
08H	读字符/属性	12H	选择可选例程
09H	显示字符/属性	13H	显示字符串
0AH	显示字符	1BH	返回视频信息

9.2 视频系统的构成

流行的(或曾经流行的)视频适配器包括 MDA(monochrome display adapter)、CGA(color graphics adapter)、EGA(enhanced graphics adapter)和 VGA(video graphics array)。VGA 及其超 VGA 的后继型号取代了 CGA 和 EGA 视频适配器。因为 VGA 系统的流行，本书只对它的特性进行说明。

视频系统的基本组成是监视器、视频显示区、视频 BIOS 以及视频控制器。其他集成的装置包括字符发生器、模式控制器、视频信号发生器和属性解码器。图 9-1 表明了它们之间的关系。

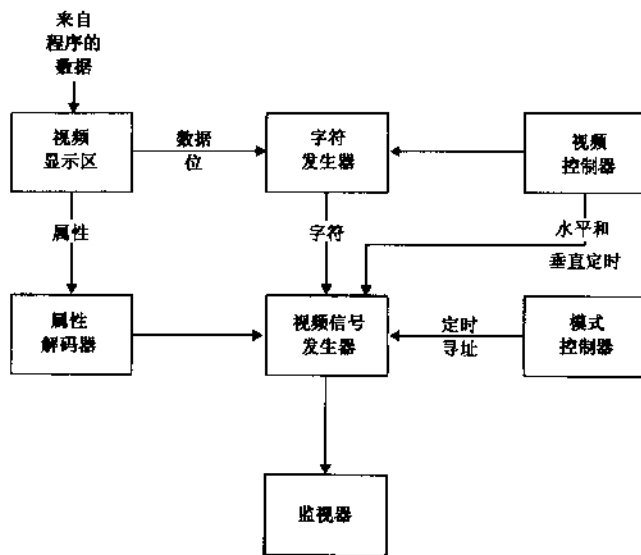


图 9-1 视频系统的组成

1. 监视器。监视器的屏幕由一组称作光栅的细密的水平线组成。每条线包含上百个称作像素的点，它由 3 个荧光点组成，每个点对应一个三原色：红、绿和蓝。

3 个电子束激活像素的 3 种颜色。电子束从屏幕的左上角开始连续地从左到右扫描每一行，改变电子束的强度来调整像素的亮度和颜色。红、绿、蓝及其亮度的组合，形成了各种颜色和阴影。

2. 视频显示区。程序中的数据——对文本方式是字符码，对图形方式是像素值——传送到 RAM 中的视频显示区(或缓冲区)，既可以通过 INT 操作，也可以通过直接传送的方法。数据最终在屏幕上显示出来之前，要经过相当复杂的转换。视频区的起始地址取决于视频适配器的类型和所选择的显示方式。处理屏幕显示的中断直接将数据传输到这个区域。下面是几个主要的视频方式所对应的起始段地址：

- A000:[0] 用于文本方式下的字形描述器以及高分辨率图形显示方式 0DH 到 13H。
- B000:[0] 单色文本方式 07H。
- B800:[0] 文本和图形方式 00H 到 06H。

视频电路连续不断地扫描视频区中的数据，因此屏幕也不断地刷新。视频区中的数据可以是 ASCII 文本(包括字母数字的)或图形格式。在文本方式下，视频区的每个字符需要两个字节：一个字节是字符，紧接着是一个决定字符颜色和亮度的属性字节。在图形方式下，视频区包含有一组决定每个像素颜色的位。

视频显示区允许按页存储数据。一页存储一个整屏的数据，页编号从 0 到 7。页号 0 是默认的，文本方式是从视频显示区的 B800[0] 开始。1 页从 B900[0] 开始，2 页在 BA00[0]，3 页在 BB00[0]，以此类推。

尽管一次只能显示一页，但你也可以对存储器中的任何页格式化。在文本方式下，要在屏幕上显示的每个字符需要两个存储器字节——一个字节存放字符，第二个字节存放它的属性。以这种方法存储，对一个 80 列和 25 行的满页字符需要 $80 \times 25 \times 2 = 4000$ 字节。实际分

配给每一页的存储器单元数是 4K 字节, 即 4096 字节, 这样一个存储块在紧接着每页后面都有 96 个字节未用。

3. 视频控制器。视频控制器产生水平和垂直定时信号。它也保留一个指示视频显示区当前位置的计数器并对它增量。该计数器告诉视频电路当前要存取、解码并送到监视器的数据。控制器必须与定时信号同步传输数据。

紧接着水平扫描之后, 控制器发出一个垂直同步信号, 使监视器执行从屏幕顶部左角开始的垂直扫描。执行过水平和垂直扫描后, 在屏幕的四周形成了一个边界(过扫描区)。

视频控制器的其他任务有处理光标的大小和位置, 并选择要显示的页。控制器也有一定数量的寄存器, 程序可对这些寄存器的内容进行读写。

如图 9-1 所示的 ASCII 字符(或字母数字)发生器将来自视频显示区的 ASCII 代码转换成组成字符的点阵, 属性解码器将来自视频显示区的属性字节转化为决定字符颜色的信号。

4. 视频 BIOS。视频 BIOS 对视频适配器来说起到一个接口的作用, 它包含像设置光标和显示字符这样的例程。视频 RAM BIOS 支持两个视频数据区:

(1) 40:[49H] 包含当前显示方式, 列数以及视频显示区大小等数据。

(2) 40:[84H] 包含行数和字符高度等数据。

关于视频 BIOS 的详细说明请见第 24 章。VGA 适配器的视频 ROM BIOS 例程起始于 C000:[0]。

9.3 视频方式

视频方式确定文本或图形、彩色或单色、屏幕分辨率以及颜色数等显示要素, BIOS INT 10H 的功能 00H 就是当前执行程序用来初始化显示方式, 或在文本方式和图形方式之间进行切换。设置显示方式也能清屏。例如, 显示方式 3 提供的是 25 行×80 列、彩色文本方式和 720×400 点阵的屏幕分辨率。也可以使用 INT 10H 的功能 0FH, 在 AL 中返回当前显示方式。这两种功能在后面都会讲到。

文本(或字母数字)方式用于显示 256 个 ASCII 字符。除彩色方式不支持下划线属性外, 彩色和单色的处理方式相类似。下表列出的是通用文本方式, 左边是显示方式的编号:

显示方式	显 示					
	行×列	类型	显示区	页	分辨率	颜色数
00	25×40	彩色	B800	0—7	360×400	16
01	25×40	彩色	B800	0—7	360×400	16
02	25×80	彩色	B800	0—3	720×400	16
03	25×80	彩色	B800	0—3	720×400	16
07	25×80	单色	B000	0	720×400	

- 文本方式 00 和 01:40 列格式; 虽然最初是为 CGA 设计的, 但是在 VGA 系统上也可以工作, 而且是完全等同的。
- 文本方式 02 和 03:80 列格式; 虽然最初是为 CGA 设计的, 但是在 VGA 系统上也可

以工作，而且是完全等同的。

- 文本方式 07(单色)：标准的单色方式。

图形方式将在后面的“使用图形方式”一节中介绍。

9.4 属 性

文本方式中的属性字节决定每一个显示字符的特性。当程序设置了一个属性时，它将保持这种设置，也就是说，之后所显示的字符全部都具有相同的属性，直到另一个操作改变了属性。使用 INT 10H 功能可以产生一个屏幕属性并且执行一些动作，如上卷或下卷、读属性或字符、显示属性或字符等。用 DEBUG 命令 D B800:0 观察视频显示区，可以看到 1 字节的字符之后，紧接着的是 1 字节的属性。

属性字节格式如下：

属性： 位：	背景				前景			
	BL	R	G	B	I	R	G	B
	7	6	5	4	3	2	1	0

字母 R、G 和 B 表示对应位分别是红、绿和蓝，每种颜色都是这三原色合成的。

- 位 7(BL) 设置闪烁(可能无效)
- 位 6—4 确定字符的背景颜色
- 位 3(I) 设置正常亮度(若为 0)或高亮度(若为 1)
- 位 2—0 确定字符的前景颜色

背景可以显示 8 种颜色中的一种，前景能显示 16 种颜色中的一种。闪烁和亮度只应用于前景，但是可以利用 INT 10H 的功能 10H 使闪烁特性无效，并使前景显示 16 种颜色。边界也可以选择 16 种颜色中的一种。

在属性字节中可以将 3 个基本视频颜色红(R)、绿(G)和蓝(B)组合形成 8 种颜色(包括黑色和白色)，设置亮度(下列表格中的 I)总共可形成 16 种颜色。

颜色	I	R	G	B	HEX	颜色	I	R	G	B	HEX
黑	0	0	0	0	0	灰	1	0	0	0	8
蓝	0	0	0	1	1	浅蓝	1	0	0	1	9
绿	0	0	1	0	2	浅绿	1	0	1	0	A
方	0	0	1	1	3	浅青	1	0	1	1	B
红	0	1	0	0	4	浅红	1	1	0	0	C
品红	0	1	0	1	5	浅品红	1	1	0	1	D
棕	0	1	1	0	6	黄	1	1	1	0	E
白	0	1	1	1	7	亮白	1	1	1	1	F

如果背景颜色和前景颜色是相同的，显示的字符是不可见的。你也可以使用属性字节让前景字符闪烁。视频系统实现闪烁的方法是：大约每两秒钟就用背景的颜色来代替前景的颜色，这样使正常字符和空白字符交替地显示。

下表是一些典型的属性，BL 的意思是闪烁：

背景	前景	背景					前景			
		BL	R	G	B	I	R	G	B	HEX
黑	蓝	0	0	0	0	0	0	0	1	01
蓝	红	0	0	0	1	0	1	0	0	14
绿	青	0	0	1	0	0	0	1	1	23
白	浅品红	0	1	1	1	1	1	0	1	7D
绿	灰(闪烁)	1	0	1	0	1	0	0	0	A8

对单色监视器，除了位 0 是设置下划线属性外，属性字节的用法和彩色监视器一样。为了指定属性，可以如下例组合各个位：

- 正常显示(黑，白) 0000 0111 (07H)
- 反相显示(白，黑) 0111 0000 (70H)

属性字节 4 位的值与控制器的彩色平面使能寄存器(Color Plane Enable register)0-3 位中的一位有关系，彩色平面使能寄存器依次指定 16 个调色板寄存器中的一个，而调色板寄存器的 0-5 位与 6 个 RGB 信号(3 个正常信号和 3 个强信号)有关。调色板寄存器中的位值指定 256 个 DAC(数模转换)颜色寄存器中的一个，颜色寄存器确定显示的颜色。

为每个字符选择属性就可以生成各种颜色。也可以修改任一个或所有调色板寄存器中默认的颜色，这可以通过后面要讲到的 INT 10H 的功能 10H 完成。

属性保持其设置直到另外的操作改变了它。设置属性的 INT 10H 功能(后面将会解释)有：

- 06H 上卷屏幕
- 07H 下卷屏幕
- 09H 显示字符及属性
- 13H 显示字符串

下例是使用 INT 10H 操作的功能 09H 在蓝色背景下显示 12 个棕色的、闪烁的星号：

```

MOV     AH, 09H           ; 请求显示功能
MOV     AL, '*'           ; 星号
MOV     BH, 00H           ; 0 页
MOV     BL, 1EH           ; 颜色属性(00011110)
MOV     CX, 12            ; 连续 12 个字符
INT     10H               ; 调用中断服务

```

你可以利用 DEBUG 来检验这个例子，并尝试用其他颜色的组合。

9.5 BIOS INT 10H 操作

INT 10H 支持很多服务例程(通过 AH 中的功能码选用)来完成视频操作。为了得到返回值，INT 操作要保存 BX、CX、DX、DI、SI、DS、ES 和 BP 的内容。一些功能还可以返回或设置在 0040:nn 的 BIOS 视频数据区某些字段中的某些位，这些将在第 24 章中讲到。

INT 10H 总是试图执行你放在它那里的任何程序，而且不返回状态代码或错误标志。所

以要特别小心选择 INT 10H 中合适的功能代码；虽然你不能因此引起任何永久性的损害，但是一个错误可能引起屏幕一片空白，这样你就不得不重新启动系统。

下面几节将描述 INT 10H 功能。

9.5.1 INT 10H 的功能 00H：设置显示方式

这个功能的目的是设置显示方式。在 AH 中装入功能代码(00H)，并在 AL 中装入所要求的显示方式。下例为在任何类型的彩色监视器上设置标准彩色文本的显示方式：

```
MOV    AH, 00H        ; 请求设置显示方式
MOV    AL, 03H        ; 标准彩色文本
INT    10H            ; 调用中断服务
```

该操作没有返回值。它能清屏，也可以不用这个功能而是采用 MOV AL, 83H 将显示方式的 7 位设置为 1。

9.5.2 INT 10H 的功能 01H：设置光标大小

光标不是 ASCII 字符集中的字符，它只存在于文本方式中。视频控制器使用专门的 INT 10H 操作处理光标大小和位置。彩色 VGA 默认的光标大小是顶部为 13，底部为 14(对单色显示是 6:7)。功能 01H 垂直地调整光标大小，设置如下的寄存器：

- CH(位 4-0)=光标顶部(扫描线开始位置)
- CL(位 4-0)=光标底部(扫描线结束位置)

下面的代码将光标放大到最大尺寸(0:14)：

```
MOV    AH, 01H        ; 请求设置光标大小功能
MOV    CH, 00         ; 开始的扫描行
MOV    CL, 14         ; 结束的扫描行
INT    10H            ; 调用中断服务
```

该操作没有返回值，现在光标是一个闪烁的实心长方形。你可以在规定的范围内调整光标的大小，例如 04:08，03:10 等等。把大小值设置为 20H 将看不见光标：MOV CX, 20H。光标保持这些属性直到另外的操作改变了它们。

9.5.3 INT 10H 的功能 02H：设置光标位置

该操作在文本方式或图形方式下，按照行:列坐标在屏幕上的任何位置设置光标(功能 13H 也设置光标)。该操作要设置这些寄存器：BH=页号(默认页号为 0)，DH=行，DL=列。下例在第 12 行，第 30 列，0 页设置光标：

```
MOV    AH, 02H        ; 请求设置光标
MOV    BH, 00         ; 页号 0 (正常的)
MOV    DH, 12         ; 第 12 行
MOV    DL, 30         ; 第 30 列
INT    10H            ; 调用中断服务
```


每页的光标位置都独立于其他页的光标。该操作没有返回值。虽然在图形方式下光标是看不见的，你仍然可以设置它。

9.5.4 INT 10H 的功能 03H：返回光标状态

在文本方式或图形方式下，使用功能 03H 可以确定光标所在行、列以及大小，特别是在程序临时使用屏幕的情况下，就必须保存和重新设置原先的屏幕。就像功能 02H 一样，在 BH 中设置页号：

```
MOV    AH, 03H    ; 请求读光标位置
MOV    BH, 00     ; 页号 0 (正常)
INT     10H       ; 调用中断服务
```

该操作不改变 AX 和 BX 并返回以下的值：

```
CH=开始的扫描行    CL=结束的扫描行
DH=行              DL=列
```

下面的例子使用功能 03H 来读光标并确定它的位置和大小，然后使用功能 02H 前移光标到屏幕上的下一列：

```
MOV    AH, 03H    ; 请求读光标位置
MOV    BH, 00     ; 0 页
INT     10H       ; 在 DL 中返回列号
MOV    AH, 02H    ; 请求设置光标
INC     DL        ; 在下一列
INT     10H       ; 调用中断服务
```

9.5.5 INT 10H 的功能 05H：选择活动页

在文本或图形方式下，功能 05H 用来选择要显示的页。你可以建立不同的页并要求在这些页之间切换。该操作就是一个简单的功能请求，没有返回值：

```
MOV    AH, 05H    ; 请求活动页
MOV    AL, page#  ; 页号
INT     10H       ; 调用中断服务
```

9.5.6 INT 10H 的功能 06H：上卷屏幕

在文本或图形方式下，该操作在屏幕的指定区域(活动显示页)执行向上卷动若干行的功能。正显示的行卷动出屏幕顶部，空白行出现在屏幕底部。

你已经使用过第 8 章介绍的功能 06H，它设置 AL 为 0 使得整个屏幕上卷，其效果是清屏。在 AL 中设置一个非 0 的值将引起上卷若干行。设置下列寄存器：

```
AL=上卷行数 (上卷全屏为 00)    CX=起始行:列
BH=属性值或像素值              DX=结束行:列
```

下例在文本方式下设置彩色属性并将全屏上卷一行：

```
MOV  AX, 601H          ; 请求上卷一行(文本方式)
MOV  BH, 61H           ; 棕色背景, 蓝色前景
MOV  CX, 0000          ; 从 00: 00 到
MOV  DX, 184FH         ; 24: 79(全屏)
INT  10H               ; 调用中断服务
```

该操作没有返回值。上卷一行的标准方法如下：

1. 设置光标的行位置，定义一个有名称的项，例如 ROW，并把它初始化为 0。
2. 显示一行并把光标设置到下一行。
3. 测试 ROW 是否接近屏幕底部(CMP ROW, 22)。
4. 如果已接近底部，则卷动一行，并用 ROW 的值来设置光标，然后把 ROW 清 0。
5. 如果还未接近底部，ROW 增量(INC ROW)。

CX: DX 寄存器允许卷动屏幕的任何部分。要注意随着 CX:DX 中的距离来调整 AL 的值，特别是涉及到部分屏幕时。下面的指令序列建立了一个 7 行×30 列的窗口(使用它自己的属性)，窗口坐标为左上角 12: 25，右上角 12: 54，左下角 18:25，右下角 18:54：

```
MOV  AX, 0607H          ; 请求卷动 7 行(文本方式)
MOV  BH, 30H            ; 青色背景, 黑色前景
MOV  CX, 0C19H          ; 从第 12 行, 第 25 列到
MOV  DX, 1236H          ; 第 18 行, 第 54 列(窗口)
INT  10H               ; 调用中断服务
```

这个例子指定了上卷 7 行，这个值与 12 行到 18 行所包括的距离是相同的，因此只有这个窗口被清屏。这是一个最常见的做法，建立一个窗口上卷(或清除)它所有的行，也可以说是连续的一次上卷一行。因为窗口的属性保持它的设置直到另一个操作改变了这个属性，因此，你可以同时对不同的窗口设置不同的属性。

在图形方式下，在 BH 中设置像素值而不是属性。下例设置若干行为红色：

```
MOV  AX, 060FH          ; 请求卷动 15 行(图形方式)
MOV  BH, 0100B          ; 像素值
MOV  CX, 0A00H          ; 从第 10 行, 第 0 列到
MOV  DX, 184FH          ; 第 24 行, 第 79 列
INT  10H               ; 调用中断服务
```

9.5.7 INT 10H 的功能 07H：下卷屏幕

在文本和图形方式下，下卷屏幕使得屏幕底部的行卷出，而在顶部出现空白行。除了该操作是下卷外，它所做的工作与上卷功能 06H 一样。设置下列寄存器：

```
AL=行数(下卷全屏为 00)      CX=起始行:列
BH=属性值或像素值           DX=结束行:列
```

9.5.8 INT 10H 的功能 08H：读光标位置的字符和属性

功能 08H 能在文本方式和图形方式下从视频显示区读出字符和它的属性。光标的位置决

定了要读出的字符。在 BH 中设置页号，如下所示：

```
MOV    AH, 08H        ; 请求读字符/属性
MOV    BH, 00         ; 页号 0 (正常)
INT    10H            ; 调用中断服务
```

该操作将字符传送到 AL，属性传送到 AH。在图形方式下，该操作返回 00H 说明读出的是非 ASCII 字符。因为该操作一次只读一个字符，所以必须编写一个循环代码来连续地读字符。

9.5.9 INT 10H 的功能 09H：在光标位置显示字符和属性

这个很有用的操作按照给定的属性显示一定数量的字符。光标的位置决定了字符显示在什么地方。设置这些寄存器：

```
AL=ASCII 字符      BL=属性或像素值
BH=页号            CX=计数值
```

CX 中的计数值指定了该操作重复显示 AL 中字符的次数。下例在文本方式下设置彩色属性，并显示 60 次“笑脸”(01H)：

```
MOV    AH, 09H        ; 请求显示 (文本方式)
MOV    AL, 01H        ; 要显示的笑脸
MOV    BH, 0          ; 页号 0
MOV    BL, 16H        ; 蓝色背景，棕色前景
MOV    CX, 60         ; 重复字符数
INT    10H            ; 调用中断服务
```

该操作不前移光标，也不对响铃(Bell)、回车、换行或 Tab 符产生反应，而是试图把它们当作 ASCII 字符来显示。在文本方式下，当显示超过了最右边一列，操作自动地继续在下一行的 0 列开始显示。

对图形方式，使用 BL 定义前景颜色。如果第 7 位是 0，定义的颜色代替已有的像素颜色；如果第 7 位是 1，定义的颜色与它们组合(异或)。

下例在图形方式下显示 10 个心形符：

```
MOV    AH, 09H        ; 请求显示 (图形方式)
MOV    AL, 03H        ; 心形符 (要显示的字符)
MOV    BH, 00         ; 页号 0
MOV    BL, 04         ; 像素值
MOV    CX, 10         ; 10 次
INT    10H            ; 调用中断服务
```

显示不同字符的申请见功能 13H。

9.5.10 INT 10H 的功能 0AH：在光标位置显示字符

功能 0AH 和 09H 唯一的区别是：功能 09H 设置属性，而功能 0AH 使用当前的属性值。下面是功能 0AH 的代码：

```

MOV    AH, 0AH           ; 请求显示
MOV    AL, char           ; 要显示的字符
MOV    BH, page#         ; 页号 (0=正常)
MOV    BL, value          ; 像素值 (仅限图形方式)
MOV    CX, repetition    ; 字符重复次数
INT    10H               ; 调用中断服务

```

该操作没有返回值。

9.5.11 INT 10H 的功能 0BH: 设置彩色调色板

在图形方式下利用这个功能来设置调色板。BH 中的值(00 或 01)确定 BL 的用途:

- BH=00 选择背景颜色, 在 BL 的 0-3 位含有颜色值(16 种颜色之一):

```

MOV    AH, 0BH           ; 请求
MOV    BH, 00            ; 背景
MOV    BL, 04            ; 红色 (选项是 00-0FH)
INT    10H               ; 调用中断服务

```

- BH=01 选择图形的调色板, BL 含有调色板(0 或 1):

```

MOV    AH, 0BH           ; 请求彩色
MOV    BH, 01            ; 选择调色板
MOV    BL, 00            ; 调色板 0 (绿, 红, 棕)
INT    10H               ; 调用中断服务

```

该操作在颜色选择寄存器中存储颜色值, 并修改 BIOS 视频显示区 40:[66]中的调色板值。

一旦设置了调色板, 它将保持这个设置, 但是当你改变调色板时, 整个屏幕就会改变颜色。

9.5.12 INT 10H 的功能 0CH: 写像素点

功能 0CH 用来在图形方式下显示所选择的颜色(背景和调色板)。设置这些寄存器:

```

AL=像素的颜色      CX=列
BH=页号            DX=行

```

列号或行号的最小值是 0, 最大值取决于显示方式。下例将像素设置在第 200 列, 第 50 行:

```

MOV    AH, 0CH           ; 请求写像素点
MOV    AL, 03            ; 像素颜色
MOV    BH, 0             ; 页号 0
MOV    CX, 200           ; 水平 x-坐标 (列)
MOV    DX, 50            ; 垂直 y-坐标 (行)
INT    10H               ; 调用中断服务

```

一种例外情况使 AL 的值成为像素的行值: 在图形方式下除 04 功能外, 设置 AL 的 7 位为 1 会引起 AL 的值与视频显示区中的值进行异或操作。

9.5.13 INT 10H 的功能 0DH: 读像素点

这个操作与功能 0CH 相反, 它读取一个像素点以确定它的颜色。在 BH 中设置页号, CX 中设置列号, DX 中设置行号。列号和行号的最小值是 0, 最大值取决于显示方式:

```
MOV    AH, 0DH          ; 请求读像素点
MOV    BH, 0            ; 页号 0
MOV    CX, 80           ; 水平 x-坐标
MOV    DX, 110          ; 垂直 y-坐标
INT     10H             ; 调用中断服务
```

该操作在 AL 中返回像素值。

9.5.14 INT 10H 的功能 0EH: 以打字方式显示

在文本和图形方式下, 该操作把监视器当作一个终端来完成简单的显示, 用法如下:

```
MOV    AH, 0EH          ; 请求显示
MOV    AL, char          ; 要显示的字符
MOV    BL, color         ; 前景颜色 (图形方式)
INT     10H             ; 调用中断服务
```

退格(08H)、响铃(07H)、回车(0DH)和换行(0AH)是屏幕显示格式的命令, 但是 Tab(09H)不起作用。该操作自动地前移光标, 自动换行到下一行显示字符, 自动卷屏并保持当前的屏幕属性。该操作没有返回值。

9.5.15 INT 10H 的功能 0FH: 取当前显示方式

可以使用这个功能来确定当前显示方式。下面是一个例子:

```
MOV    AH, 0FH          ; 请求显示方式
INT     10H             ; 调用中断服务
CMP    AL, 03           ; 如是方式 03
JE      ...             ; 则跳转
```

该操作从 BIOS 视频数据区返回这些值: AL=当前显示方式, AH=屏幕列数, BH=活动显示页。

9.5.16 程序: 显示 ASCII 字符集

图 8-1 的程序使用 INT 21H 的功能 09H 来显示 ASCII 字符集, 但是它不能显示退格、响铃、回车和换行等控制符。图 9-2 是修改后的程序, 它使用 INT 10H 来显示全部的字符集, 所使用的功能如下:

```
0FH      取当前显示方式并保存它。
```

- 00H 本程序设置显示方式 03, 在退出程序之前恢复原先的显示方式。
 08H 为了用功能 06H 清屏, 先读出当前光标位置的属性。
 06H 上卷屏幕以清除整个屏幕, 使用刚才读出来的属性。同时建立一个 16 行的, 棕色前景和蓝色背景的窗口来显示字符。
 02H 设置光标的初始位置, 并为每个要显示的字符前移光标。
 0AH 在当前光标位置上显示每个字符, 包括控制字符。

要显示的字符在一个 16 列和 16 行的栅格内。编写这个程序和本书的其他程序一样, 要求清晰而不是处理速度。你可以修改这个程序使它运行的更快, 例如, 处理行、列以及 ASCII 字符值使用寄存器。因为 INT 10H 只破坏 AX 的内容, 所以不必再装入其他寄存器的值。然而要使这个程序明显地加快运行速度是可以的, 但程序要损失一些清晰度。

```

TITLE      A09BIOAS [EXE] 利用 INT 10H 显示 ASCII 字符集
.MODEL     SMALL
.STACK    64
.DATA
CHAR_CTR  DB  00          ; ASCII 字符的计数器
COL        DB  24          ; 屏幕的列
ROW        DB  04          ; 屏幕的行
MODE       DB  ?           ; 显示方式
.286
;-----
.CODE
A10MAIN    PROC    NEAR
MOV        AX,@DATA      ; 初始化段寄存器
MOV        DS,AX
MOV        ES,AX
CALL       B10MODE        ; 获取/设置显示方式
CALL       C10CLEAR       ; 清屏

A20:
CALL       D10CURSOR      ; 设置光标
CALL       E10DISPLY      ; 显示字符
CMP        CHAR_CTR,0FFH  ; 最后一个显示字符?
JE         A30            ; 是, 退出
INC        CHAR_CTR       ; 字符计数器加1
ADD        COL,02         ; 列号加1
CMP        COL,56         ; 在最后一行?
JNE        A20            ; 否, 跳转
INC        ROW            ; 是, 则行号加1
MOV        COL,24         ; 并重新设置列
JMP        A20

A30:
MOV        AH,10H         ; 请求从键盘
INT        16H            ; 获取字符
MOV        AH,00H         ; 请求重置显示方式
MOV        AL,MODE        ; 为初始值
INT        10H
MOV        AX,4C00H       ; 处理结束
INT        21H

A10MAIN    ENDP
;
;----- 获取并保存当前显示方式, 设置新的显示方式: -----
B10MODE    PROC    NEAR
MOV        AH,0FH         ; 请求获取显示方式
INT        10H
MOV        MODE,AL        ; 保存显示方式
MOV        AH,00H         ; 请求设置新的显示方式
MOV        AL,03          ; 标准彩色
INT        10H
RET

B10MODE    ENDP
;
;----- 清屏, 创建窗口, 设置属性: -----
C10CLEAR   PROC    NEAR
PUSHA
MOV        AH,06H         ; 保存通用寄存器
INT        10H            ; 请求取当前属性
MOV        BH,AH          ; 存入 AH
MOV        BH,AH          ; 属性移入 BH

```

图 9-2 INT 10H 显示 ASCII 字符集

```

MOV     AX, 0600H      ; 滚动整个屏幕
MOV     CX, 0000      ; 左上角位置
MOV     DX, 184FH     ; 右下角位置
INT     10H
MOV     AX, 0610H     ; 创建16行的窗口
MOV     BH, 16H       ; 蓝色背景, 棕色前景
MOV     CX, 0418H     ; 左上角04:24
MOV     DX, 1336H     ; 右下角19:54
INT     10H
POPA
RET      ; 恢复寄存器

C10CLEAR ENDP
;
; ----- 设置光标的行和列: -----
D10CURSOR PROC NEAR
PUSH    AH      ; 保存通用寄存器
MOV     AH, 02H  ; 请求设置光标
MOV     BH, 00   ; 0 页 (正常属性)
MOV     DH, ROW  ; 新的行
MOV     DL, COL  ; 新的列
INT     10H
POPA
RET      ; 恢复寄存器
D10CURSOR ENDP
;
; ----- 一次显示一个 ASCII 字符: -----
E10DISPLY PROC NEAR
PUSH    AH      ; 保存通用寄存器
MOV     AH, 0AH  ; 请求显示
MOV     AL, CHAR_CTR ; ASCII 字符
MOV     BH, 00   ; 0 页
MOV     CX, 01   ; 一个字符
INT     10H
POPA
RET      ; 恢复寄存器
E10DISPLY ENDP
END      A10MAIN

```

图 9-2 续

9.5.17 INT 10H 的功能 10H: 存取调色板寄存器和视频 DAC

这个操作提供了若干个有关读取和修改调色板寄存器、过扫描(边界)寄存器以及视频 DAC 的功能。在 AL 中装入子功能以指定一个执行例程。

1. 子功能 00H: 修改调色板寄存器。通过选择 16 个调色板寄存器中的一个可以改变显示的颜色, 在 BH 中装入颜色值, 在 BL 中装入调色板寄存器编号(00-0FH):

```

MOV     AX, 1000H      ; 请求修改调色板寄存器
MOV     BH, 02         ; 新的颜色(绿色)
MOV     BL, 01         ; 调色板寄存器
INT     10H            ; 调用中断服务

```

2. 子功能 01H: 修改边界颜色。默认边界颜色是黑色。为了改变边界颜色, 在 BH 中放入新的颜色并请求这个操作:

```

MOV     AX, 1001H      ; 请求修改边界颜色
MOV     BH, 02         ; 新颜色(绿色)
INT     10H            ; 调用中断服务

```

3. 子功能 03H: 选择背景亮度。该操作可以允许或禁止闪烁属性, 在 BL 中装入代码(00H=禁止, 01H=允许)。这个操作存取属性控制器的方式控制寄存器。由于禁止闪烁, 背景颜色

可选用所有的 16 个调色板寄存器而不是 8 个调色板寄存器。

```
MOV    AX, 1003H          ; 请求
MOV    BL, 00H            ; 禁止闪烁
INT     10H               ; 调用中断服务
```

4. 子功能 07H: 读调色板寄存器。该操作允许将确定的颜色码保存在 16 个调色板寄存器中的任何一个中。BL 中是请求的寄存器号:

```
MOV    AX, 1007H          ; 请求调色板寄存器中的
MOV    BL, register       ; 颜色码
INT     10H               ; 调用中断服务
```

该操作在 BH 中返回颜色码。参见子功能 09H。

5. 子功能 08H: 读过扫描寄存器。该操作返回过扫描(边界)寄存器中的当前的颜色码:

```
MOV    AX, 1008H          ; 读过扫描寄存器
INT     10H               ; 调用中断服务
```

该操作在 BH 中返回颜色码。

6. 子功能 09H: 读调色板寄存器值表。该操作返回所有当前的调色板和过扫描寄存器的值, 并把它们存入一个 17 字节的表中。你可以定义这个表并把它地址送入 ES: DX。下例假定 ES 已包含有适当的段地址:

```
REGTABLE DB 17 DUP(?)      ; 17 字节表
...
MOV     AX, 1009H          ; 读调色板寄存器值
LEA     DX, REGTABLE      ; 放入表中 (ES: DX)
INT     10H               ; 调用中断服务
```

该操作把调色板寄存器的内容返回到前 16 个字节中, 过扫描寄存器放入第 17 个字节中。典型的默认值是 00 01 02 03 04 05 14 07 38 39 3A 3B 3C 3D 3E 3F 00。这最后一个值 00 表示边界颜色是黑色。

7. 子功能 10H: 修改 DAC 颜色寄存器。视频 DAC(digital-to-analog converter)包含有 256 个 3 字节的颜色寄存器:

```
00-0FH    默认的 CGA 彩色系列
10-1FH    增加亮度的灰度色标
20-67H    3 组中的第 1 组: 高亮度蓝、红、绿
68-AFH    3 组中的第 2 组: 中亮度蓝、红、绿
B0-F7H    3 组中的第 3 组: 低亮度蓝、红、绿
F8-FFH    黑色
```

在 20、68 和 B0 的 3 组, 每组都由按饱和度递减顺序排列的 3 种颜色组成。子功能 10H 能用颜色值修改任一 DAC 寄存器:

```
MOV     AX, 1010H          ; 修改 DAC 颜色寄存器
MOV     BX, register       ; DAC 寄存器号
MOV     CH, green          ; 绿的颜色值
MOV     CL, blue           ; 蓝的颜色值
MOV     DH, red            ; 红的颜色值
INT     10H               ; 调用中断服务
```


该操作只使用颜色码的低6位。

8. 子功能 12H: 修改 DAC 寄存器组。该操作能用颜色值修改一组 DAC 寄存器。定义一个颜色表(每个寄存器 3 字节分别对应红、绿和蓝), 并把它的地址送入 ES: DX。下例假定 ES 已含有适当的段地址:

```

DACTABLE DB nn DUP(?)           ; 表 (每个寄存器 3 字节)
...
MOV      AX, 1012H               ; 修改 DAC 寄存器组
MOV      BX, register            ; 组内第一个 DAC 寄存器
MOV      CX, number              ; 组内 DAC 寄存器数
LEA      DX, DACTABLE            ; 放入表中 (ES: DX)
INT      10H                     ; 调用中断服务

```

9. 子功能 15H: 读视频 DAC 颜色寄存器。子功能 15H 能读取任一个 DAC 寄存器:

```

MOV      AX, 1015H               ; 读视频 DAC 颜色寄存器
MOV      BX, register            ; DAC 寄存器号
INT      10H                     ; 调用中断服务

```

该操作在 CH 中返回绿色码, 在 CL 中返回蓝色码, 在 DH 中返回红色码,

10. 子功能 17H: 读 DAC 寄存器组。该操作能读取一组 DAC 寄存器(最多 256 个)存入一个表中(对应红、绿、蓝每个寄存器 3 字节), 表地址在 ES: DX 中。下例假定 ES 已含有适当的段地址:

```

DACTABLE DB nn DUP(?)           ; 表 (每个寄存器 3 字节)
...
MOV      AX, 1017H               ; 读一组 DAC 寄存器
MOV      BX, register            ; 组内第一个 DAC 寄存器
MOV      CX, number              ; 组内 DAC 寄存器数
LEA      DX, DACTABLE            ; 表 (ES: DX)
INT      10H                     ; 调用中断服务

```

能够看到表中的返回值, 如 00 00 00(寄存器 0), 00 00 2A(寄存器 1), 00 2A 00(寄存器 2), 依此类推。

11. 子功能 1BH: 对 DAC 寄存器组执行灰度定标。该操作能对一组视频 DAC 寄存器执行灰度定标:

```

MOV      AX, 101BH               ; 执行灰度定标
MOV      BX, register            ; 组内第一个 DAC 寄存器
MOV      CX, number              ; 组内 DAC 寄存器数
INT      10H                     ; 调用中断服务

```

设置 BX=0 和 CX=10H 来做个试验, 可看到灰度定标的效果。

9.5.18 INT 10H 的功能 11H: 存取字符发生器

这个操作支持与文本和图形字符发生器有关的若干子功能, 这些功能可以用来改变显示字符的尺寸和形状。下面的图表列出了 12 个有关的子功能:

装入基于文本的字符
装入基于文本的字符和
程序视频控制器
装入基于图形的字符

用户定义表	字符尺寸 8×14	字符尺寸 8×14	字符尺寸 8×14
00H	01H	02H	04H
10H	11H	12H	14H
21H	22H	23H	24H

1. 子功能 00H, 10H 及 21H。这些操作所包括的用户定义字符是本书范围之外的内容。

2. 子功能 01H, 02H 及 04H。这些操作支持 3 种预定义尺寸的基于文本的字符。

3. 子功能 11H, 12H 及 14H(默认值)。这些操作分别与 00H、01H 和 04H 类似, 但是也可对视频控制器重新编程以处理字符矩阵的高度。使用这些功能要在 BL 中装入字符发生器的编码。下例装入 8×8 基于文本的字符:

```
MOV    AX, 1112H      ; 装入 8×8 文本字符
MOV    BL, 0          ; 表编码 0-7
INT    10H            ; 调用中断服务
```

该操作能使屏幕显示 43 行的 8×8 大小的字符。

4. 子功能 22H, 23H 及 24H。这些操作支持基于图形的字符。对这些功能, 在 BL 中要指出每屏的字符行数, 这里 1=14 行, 2=25 行, 3=43 行, 0=DL 中给定的每屏字符行数:

```
MOV    AX, 1123H      ; 装入 8×8 图形字符
MOV    BL, code        ; 0, 1, 2 或 3
MOV    DL, number      ; 当 BL=0 时的字符行数
INT    10H            ; 调用中断服务
```

5. 子功能 03H: 选择显示字符定义表。在 BL 中装入一个位串。根据串属性(亮度)的位 3 的值:

- 如果为 0, 位 0、1 和 4 指出使用 8 个 256 字符表中的一个。
- 如果为 1, 位 2、3 和 5 指出要使用的字符表。

6. 子功能 30H: 读字符发生器数据。为了取得有关当前字符发生器的数据, 把下列代码之一装入 BH:

```
0= INT 1FH 向量的内容
1= INT 43H 向量的内容
2= BIOS 8×14 字符表的地址
3= BIOS 8×8 字符表前一半的地址
4= BIOS 8×8 字符表后一半的地址
5= BIOS 9×14 交替字符表的地址
6= BIOS 8×16 字符表的地址
7= BIOS 9×16 交替字符表的地址
```

该操作返回这些值: CX=字符点阵的高度, DL=行数-1(典型的是 18H), ES:BP=字符定义表的地址。你可以利用 DEBUG(文本方式)来显示每个选项:

```
MOV    AX, 1130      ; 请求字符发生器信息
MOV    BH, 0         ; 代码 0, 1, 2, ..., 7
INT    10            ; 调用中断服务
```

```

INC    BH                ; 下一个 DAC 寄存器
JMP    100               ; 重复

```

用 DEBUG 的 D 命令来显示传送到 ES:BP 的段地址:偏移地址, 如 D ES:偏移地址。

9.5.19 INT 10H 的功能 12H: 选择可选视频例程

该操作支持许多子功能, 子功能编码在 BL 中:

- 子功能 10H: 返回视频结构信息。该操作返回:

```

BH=视频类别 (0 为彩色, 1 为单色)
BL=视频 RAM 容量 (0=64K, 1=128K, 2=192K, 3=256K+)
CH=适配器位
CL=结构开关设置

```

- 子功能 30H: 为文本方式选择扫描线。在文本方式的 0、1、2、3 和 7 方式下, 可利用这个操作来改变扫描线的垂直分辨率。在 AL 中装入扫描线数: 0=200, 1=350, 2=400。然后用功能 00H 设置显示方式:

```

MOV    AX, 1202H          ; 请求选择 400 行
MOV    BL, 30H            ; 扫描线
INT     10H               ; 调用中断服务
MOV    AX, 0003H          ; 设置文本方式 03 (720×400)
INT     10H               ; 调用中断服务

```

- 其他子功能包括允许和禁止灰度求和、光标仿真和恢复控制。

9.5.20 INT 10H 的功能 13H: 显示字符串

这是一个功能强大的操作, 它在文本或图形方式下显示任意长度的字符串, 并带有设置属性和移动光标的选项。在 ES:BP 中装入显示串的段地址:偏移地址。该操作对退格、响铃、回车和换行等控制字符起作用, 但对 Tab 不起作用。

```

MOV    AH, 13H            ; 请求显示串
MOV    AL, subfunction     ; 00, 01, 02, 或 03 (见下面的解释)
MOV    BH, page#          ; 页号
MOV    BL, attribute       ; 屏幕属性
LEA    BP, address        ; 串地址在 ES:BP
MOV    CX, length          ; 串长度
MOV    DH, row             ; 屏幕行
MOV    DL, column          ; 屏幕列
INT     10H               ; 调用中断服务

```

设置在 AL 中的 4 个子功能是:

- 00 显示串和属性; 不前移光标
- 01 显示串和属性; 前移光标
- 02 显示字符及其属性; 不前移光标
- 03 显示字符及其属性; 前移光标

子功能 02 和 03 要求显示串的每个字符之后紧接着一个属性字节。这个特点对显示带有混合属性的数据非常方便,并且可以直接从视频显示区显示。

在图形方式下,设置 BL 的 7 位为 1,可使串值与视频区的值进行 XOR (异或) 操作。

9.5.21 程序: 设置属性和卷屏

图 9-3 的程序从键盘接收并在屏幕上显示名字。为了使显示更加有趣,用反相显示(白底蓝字)提示符,接收并正常显示(蓝底白字)名字,这样在同一行的 70 列右调整为反相显示,格式如下:

```
Name? Mark Twain ... Mark Twain
|               |
0 列           70 列
```

程序由以下几个子程序组成:

- A10MAIN 提供主要的逻辑功能,即接收任意数量的键盘输入以及使用 INT 10H 的功能 10H 来禁止闪烁属性。
- B10PROMPT 为用户显示提示符以接收名字。
- C10INPUT 用 INT 21H 的功能 0AH 接收键盘输入。
- D10NAME 计算起始列使得输入的名字在显示时是右调整的。调用 E10DISPLY 向下显示直到屏幕的 23 行,然后每增加一个提示符就上卷一行。
- E10DISPLY 使用 INT 10H 的功能 13H 来设置光标并显示输入的名字。
- Q10SCOLL 处理屏幕的卷动,设置屏幕背景为灰色(1000B),它是在禁止闪烁的情况下工作的。

```

TITLE      A09NMSCR (EXE) 设置属性并卷屏
.MODEL     SMALL
.STACK     64
.DATA
PARLIST    LABEL    BYTE           ; 名字参数表
MAX_LEN    DB        20            ; 名字的最大长度
ACT_LEN    DB        ?             ; 输入名字的长度
KB_NAME    DB        20 DUP(' ')  ; 字符数

LEFT_COL   EQU       51            ; 显示的左边列
BOTT_SCRN  EQU       23            ; 显示的底边行
ATTRIB     DB        00            ; 屏幕属性
COL         DB        05            ; 屏幕的列
ROW         DB        00            ; 行
PROMPT     DB        'Name? '      ; 输入提示符
.386 ;
-----
A10MAIN    .CODE
PROC       FAR
MOV        AX,@data                ; 初始化段寄存器
MOV        DS,AX
MOV        ES,AX
MOV        AX,1003H                ; 显示闪烁
MOV        BL,00                    ; 属性
INT        10H
MOV        AL,00H                  ; 请求清除全屏
CALL       Q10SCROLL
;
A20:
MOV        COL,05                  ; 设置列为 0
CALL       B10PROMPT                ; 显示提示符

```

图 9-3 反相显示和卷屏

```

CALL C10INPUT ; 用于输入名字
CMP ACT_LEN, 00 ; 无名字输入? (指示结束)
JE A30 ; 是, 退出
CALL D10NAME ; 显示名字
JMP A20

A30: MOV AL, 00H ; 退出,
CALL Q10SCROLL ; 清屏,
MOV AX, 4C00H ; 处理结束
INT 21H

A10MAIN ENDP
;
; ----- 为用户显示提示符: -----
;
B10PROMPT PROC NEAR ; 使用BP 和 CX
MOV ATTRIB, 71H ; 设置属性
LEA BP, PROMPT ; 设置提示符地址
MOV CX, 06 ; 和长度
CALL E10DISPLY ; 显示例程
RET
B10PROMPT ENDP
;
; ----- 从键盘接收输入的名字: -----
;
C10INPUT PROC NEAR ; 使用 AH 和 DX
MOV AH, 0AH ; 请求键盘输入
LEA DX, PARLIST ;
INT 21H
RET
C10INPUT ENDP
;
; ----- 为显示名字设置参数, -----
; ----- 如接近屏幕底边则卷屏: -----
;
D10NAME PROC NEAR
PUSHA ; 保存通用寄存器
MOV AL, MAX_LEN ;
SUB AL, ACT_LEN ; 计算行首偏移
ADD AL, LEFT_COL ; 与左列相加
MOV COL, AL ; 并存值
MOV ATTRIB, 17H ; 反相显示
LEA BP, KB_NAME ; 初始化名字和
MOVZX CX, ACT_LEN ; 长度
CALL E10DISPLY ; 显示名字
CMP ROW, BOTT_SCRN ; 接近屏幕底边?
JAE D30 ; 是, 跳转
INC ROW ; 否, 增加1行
JMP D90 ; 并退出
D30: MOV AL, 01H ; 卷屏
CALL Q10SCROLL ; 1行
D90: POPA ; 恢复通用寄存器
RET
D10NAME ENDP
;
; ----- 显示字符并设置属性: -----
;
E10DISPLY PROC NEAR ; BP, CX 已输入数据
PUSHA ; 保存通用寄存器
MOV AH, 13H ; 请求显示
MOV AL, 01 ; 字符
MOV BH, 00 ; 页号
MOV BL, ATTRIB ; 属性
MOV DH, ROW ; 屏幕行
MOV DL, COL ; 和列
INT 10H
POPA ; 恢复通用寄存器
RET
E10DISPLY ENDP
;
; ----- 卷屏和设置属性: -----
;
Q10SCROLL PROC NEAR ; AL 已设置输入参数
PUSHA ; 保存通用寄存器
MOV AH, 06H ; 请求卷屏
MOV BH, 86H ; 全屏为
MOV CX, 0000 ; 灰底
MOV DX, 184FH ; 棕字

```

图 9-3 续

```

                INT     10H
                POPA
                RET     ; 恢复通用寄存器
Q10SCROLL ENDP
                END     A10MAIN

```

图 9-3 续

9.5.22 INT 10H 的功能 1AH: 视频显示的组合

该操作支持许多与组合视频子系统(例如, 彩色和单色显示器)有关的子功能。这些讨论超出了本书的范围。

9.5.23 INT 10H 的功能 1BH: 返回视频 BIOS 信息

对于这个操作, 你要为动态信息定义一个 64 字节的缓冲区, 并将缓冲区的地址送入 ES:DI。在 DEBUG 中, 该操作的代码如下:

```

MOV  AH, 1B          ; 请求视频 BIOS 信息
MOV  BX, 0           ; 0 为实现类型
MOV  DI, 110         ; 动态信息的偏移地址 (ES:DI)
INT  10              ; 调用中断服务
JMP  100

```

该操作返回下列动态信息(本例中, 地址为 ES:110):

偏移地址	字长	说 明
00H	双字	静态功能表的地址 (见下面)
04H	字节	当前显示方式
05H	字	屏幕列数(5000H=80)
07H	字	当前视频缓冲区大小(0010H=4 096)
09H	字	当前视频缓冲区起始地址
0BH	数组	16 字节数组, 保存 8 页的光标列:行
1BH	字节	光标结束扫描行
1CH	字节	光标开始扫描行
1DH	字节	当前显示页
1EH	字	CRTC 地址寄存器端口
20H	字节	显示方式寄存器的当前设置
21H	字节	当前彩色调色板
22H	字节	当前屏幕行号(19H=25)
23H	字	字符扫描行高度(1000H=16)
25H	多字节	活动的和不活动的显示组合码
27H	字	当前显示颜色数
29H	字节	显示页数
2AH	字节	扫描线(0=200, 1=350, 2=400, 3=480)

2BH	多字节	正常/高亮度的文本字符表
2DH	字节	位 0-5 指示允许/禁止条件
31H	字节	可用的视频存储器 (3=256K+)
32H	字节	位 0-5 指示活动/非活动条件

动态表的起始地址指向一个 16 字节的静态功能表。例如, 内容为 8839 00C0 意味着偏移地址为 3988H, 段地址为 C000H, 这可以利用 DEBUG 来观察表, 命令为 D C000: 3988。

偏移地址	字长	说 明
00H	字节	位 0=1 说明支持方式 0, 以此类推
01H	字节	位 0=1 说明支持方式 8, 以此类推
02H	字节	位 0=1 说明支持方式 10H, 以此类推
07H	字节	位=1, 若为文本方式支持的扫描行(位 0 为 200 行, 位 1 为 350 行, 位 2 为 400 行)
08H	字节	可显示文本字符集的最大数
09H	字节	文本字符定义表的数目
0AH	字节	位 0-3 指示不同的视频存储器容量

9.5.24 INT 10H 的功能 1CH: 保存和恢复视频状态

该操作支持返回缓冲区大小、保存请求的状态和恢复请求的状态等 3 个子功能。关于该操作的讨论超出了本书的范围。

9.6 使用图形方式

图形方式利用像素(picture elements 或 pels)来产生彩色图案。在前面我们已经知道文本方式下的属性由 4 位背景色和 4 位前景色组成。视频系统以一种类似的, 但是又与显示方式相关的方法来分辨像素, 这种方法可以用 1 到 8 位来表示一个像素。

尽管光标仍可以设置, 但设置图形方式将使光标消失。下面是通用的图形方式:

显示方式	类型	显示区	页	分辨率	彩色
04H	彩色	B800	8	320×200	4
05H	彩色	B800	8	320×200	4
06H	彩色	B800	8	640×200	2
0DH	彩色	B800	8	320×200	16
0EH	彩色	A000	4	640×200	16
0FH	单色	A000	2	640×350	1
10H	彩色	A000	2	640×350	16
11H	彩色	A000	1	640×480	2
12H	彩色	A000	1	640×480	16
13H	彩色	A000	1	320×200	256

- 图形方式 04H 和 05H。为了向上兼容，最初的 CGA 方式也用于 VGA。一个字节表示 4 个像素(每个像素两位)。两位可同时提供 2^2 ，即 4 种不同的颜色。
- 图形方式 06H。为了向上兼容，最初的 CGA 方式用于 VGA。一个字节表示 8 个像素(每个像素一位)，同时可以提供两种颜色。
- 图形方式 0DH、0EH 和 10H。为了向上兼容，最初的 EGA 方式也用于 VGA。在 16 色图形方式下，每个屏幕字符是一个 8×8 的像素矩阵。视频显示区用 32 字节的数据来表示像素值，也就是说，每像素 $4 \text{ 位} \times 8 \times 8 = 256 \text{ 位} = 32 \text{ 字节}$ 。
- 图形方式 0FH。为了向上兼容，最初的 EGA 单色方式也用于 VGA。
- 图形方式 11H 和 12H。它们是专门为 VGA 设计的。这些方式在操作上与方式 0DH 和 0EH 类似；在技术上，方式 11H 只需要一个位图来表示一个屏幕的像素。然而，在位图中 11H 可存取到两种调色板颜色，方式 12H 可存取到 16 种调色板颜色。
- 图形方式 13H。它们是专门为 VGA 设计的。这种方式也使用像素位图，但是每个像素由视频位图的一个字节来表示。8 位提供 2^8 ，即 256 种不同的颜色。

使用 BIOS INT 10H 的功能 00H 来设置图形方式，如下例所示：

```

MOV    AH, 00H           ; 请求设置方式
MOV    AL, 0CH           ; 彩色图形
INT     10H              ; 调用中断服务

```

在图形方式下，ROM 只包含了第一个(底部)128 ASCII 字符的点阵图。INT 1FH 提供可访问的 1K 的存储区，其中定义了顶部的 128 个 ASCII 字符，每个字符 8 字节。

9.6.1 程序：使用图形方式的功能

图 9-4 的程序使用的图形方式功能如下：

- 0FH 取得并保存原先设定的显示方式。
- 00H 设置图形方式 12H。
- 0BH 请求一个彩色调色板
- 06H 滚动屏幕并设置 5 行为黄色。
- 13H 设置光标和属性并显示一串字符。
- 0AH 显示一个字符并重复若干次。

在程序末尾等待用户按动一个键，然后重新设置为原来的显示方式。

```

TITLE      A09GRFX1 (EXE) 显示的图形功能
.MODEL     SMALL
.STACK     64
.DATA
STRING     DB      '1234567890'
.286 ;
-----
.CODE
A10MAIN    PROC     FAR
MOV        AX,@data      ; 设立可寻址寄存器
MOV        DS,AX         ;
MOV        ES,AX
MOV        AH,0FH        ; 获取初始显示
INT         10H          ; 方式
PUSH       AX            ; 并保存

```

图 9-4 视频显示的图形功能


```

CALL B10MODE ;设置图形方式
CALL C10SCROLL ;滚屏
CALL D10STRING ;显示功能 13H
CALL E10DISPLY ;显示功能 0AH
MOV AH,10H ;请求键盘输入
INT 16H ;
POP AX ;恢复初始
MOV AH,00H ; 显示方式
INT 10H ; (在AL中)
MOV AX,4C00H ;处理结束
INT 21H ;
A10MAIN ENDP
;
; 设置图形方式并请求调色板:
B10MODE PROC NEAR ;使用 AX 和 BX
MOV AH,00H ;请求图形方式
MOV AL,12H ;640列 × 480行
INT 10H
MOV AH,0BH ;请求彩色调色板
MOV BH,00 ;背景
MOV BL,07H ;灰色
INT 10H ;
RET
B10MODE ENDP
;
; 滚屏、设置属性:
C10SCROLL PROC NEAR
PUSHAX ;保存通用寄存器
MOV AX,0605H ;请求滚动5行
MOV BH,1110B ;黄色
MOV CX,0000H ;从行:列到
MOV DX,044FH ;行:列
INT 10H
POPA ;恢复寄存器
RET
C10SCROLL ENDP
;
; 显示串, 设置属性和光标:
D10STRING PROC NEAR
PUSHAX ;保存通用寄存器
MOV AX,1301H ;请求显示
MOV BX,00021H ;页:属性
LEA BP,STRING ;字符串
MOV CX,10 ;长度
MOV DX,0815H ;行:列
INT 10H ;
POPA ;恢复寄存器
RET
D10STRING ENDP
;
; 重复显示字符:
E10DISPLY PROC NEAR
PUSHAX ;保存通用寄存器
MOV AX,0A01H ;请求显示
MOV BH,00 ;笑脸
MOV BL,0100B ;红色
MOV CX,10 ;10次
INT 10H ;
POPA ;恢复寄存器
RET
E10DISPLY ENDP
END A10MAIN

```

图 9-4 续

9.6.2 像素举例

作为一个简单的例子,显示方式 04H 提供了 200 行的 320 个像素。在这种方式下,每个字节表示 4 个像素(即每个像素两位),编号为 0-3,如下所示:

字节:	C1 C0	C1 C0	C1 C0	C1 C0
像素:	0	1	2	3

在任何给定的时间,有4种可用的颜色,编号为0-3。局限为4种颜色是因为2位像素值只提供4种位组合:00,01,10和11。你可以设置像素值00来选择16种可用颜色之一作为背景色。

颜色		颜色	
黑	0000	灰	1000
蓝	0001	浅蓝	1001
绿	0010	浅绿	1010
青	0011	浅青	1011
红	0100	浅红	1100
品红	0101	浅品红	1101
棕	0110	黄	1110
灰白	0111	白	1111

对于方式04H,调色板0由前景的绿色、红色和棕色组成,调色板1由前景的青色、品红和白色组成。

使用INT 10H的功能0BH来选择一种调色板和背景色。如果选择背景色为黄色及调色板0,则可用的颜色是黄、绿、红和棕。一个由像素值10101010组成的字节显示为全红。如果选择背景色为蓝色及调色板1,则可用的颜色为蓝、青、品红和白,那么一个由像素值00011011组成的字节显示蓝、青、品红和白色。

9.6.3 程序:显示图形像素

图9-5的程序包括了以下INT 10H功能显示图形:

0FH=取得原始的显示方式
00H=设置图形方式12H
0BH=选择背景色为绿色
0CH=在640列,480行显示点

实际的显示窗口为210行,512列(从第64列到第576列)。注意行和列是针对点来说的,而不是对字符。

```

TITLE      A09GRFX2 (EXE) 显示像素
.MODEL     SMALL
.STACK     64

.286
;-----
.CODE
A10MAIN    PROC    FAR
MOV        AX,@data      ; 设立可寻址寄存器
MOV        DS,AX
MOV        ES,AX
MOV        AH,0FH        ; 获得初始
INT         10H          ; 显示方式
PUSH       AX            ; 并保存

```

图9-5 图形显示像素

```

CALL B10MODE ; 设置图形方式
CALL C10DISPLY ; 显示彩色图形
MOV AH, 10H ; 请求键盘响应
INT 16H ;
POP AX ; 恢复初始显示
MOV AH, 00H ; 方式
INT 10H ; (在 AL 中)
MOV AX, 4C00H ; 处理结束
INT 21H

A10MAIN ENDP
;
; ----- 设置图形方式和调色板: -----
;
B10MODE PROC NEAR ; 使用 AX 和 BX
MOV AX, 0012H ; 请求图形方式
INT 10H ; 640 列 × 480 行
MOV AH, 0BH ; 请求彩色调色板
MOV BX, 0007H ; 背景为灰色
INT 10H
RET
B10MODE ENDP
;
; ----- 显示 210 行图形点, 512 列, 每行改变一种颜色: -----
;
C10DISPLY PROC NEAR
PUSHA ; 保存通用寄存器
MOV BX, 00 ; 设置初始页
MOV CX, 64 ; 列
MOV DX, 70 ; 和行
C20:
MOV AH, 0CH ; 请求像素点
MOV AL, BL ; 颜色
INT 10H ; 保存 BX, CX, 和 DX
INC CX ; 列加 1
CMP CX, 576 ; 在 576 列?
JNE C20 ; 否, 循环
MOV CX, 64 ; 是, 重置列
INC BL ; 改变颜色
INC DX ; 增加行
CMP DX, 280 ; 行为 280 行?
JNE C20 ; 否, 循环
POPA ; 恢复寄存器
RET ; 是, 结束
C10DISPLY ENDP
END A10MAIN

```

图 9-5 续

程序增加每一行的颜色值(即位值 0000 变为 0001, 等等), 因为只用到最右边的 4 位, 所以这些颜色每 16 行重复显示一次。显示从屏幕左边的 64 列开始到屏幕右边的 64 列结束。

在程序末尾等待用户按动一个键, 然后重新设置为最初的显示方式。你可以将程序修改为其他的图形方式。

9.7 直接视频显示

由于对某些应用来说, 视频显示经过操作系统和 BIOS 可能会明显地变慢。显示文本和图形字符最快的方法是直接将它们传输到相应的视频显示区。例如, 对显示方式 03(彩色文本), 0 页在视频区的地址是 B800[0]H。每个字符需要两个存储器字节——一个字节存储字符, 紧接着的另一个字节存储它的属性。一屏的大小为 80 列和 25 行, 则一页在视频区占用 $80 \times 25 \times 2 = 4000$ 字节。

视频显示区中的头两个字节表示的屏幕位置是 00 行和 00 列, 偏移地址为 F9EH 和 F9FH

的两个字节表示 24 行和 79 列的屏幕位置。简单移动一个字符:属性到活动页的视频区,屏幕上立即显现出字符。可以用 DEBUG 命令来检验这个特性。首先,用命令 D B800:00 来显示在 B800[0]H 的视频区。在屏幕上显示出来的是当时你键入的命令,通常是含有 20 07H(空字符,黑色背景和白色前景)的一组字节。注意,DEBUG 和你的输入会竞争同一个显示区和屏幕位置。尝试改变屏幕显示,使用这些命令在顶部行和底部行用不同的属性(25、36 和 47)来显示笑脸(01、02 和 03):

```
E B800:000 01 25 02 36 03 47
E B800:F90 01 25 02 36 03 47
```

图 9-6 的程序给出一个直接传输数据到视频显示区 B900[0]H 的例子,该地址是 1 页而不是默认的 0 页。程序使用 SEGMENT AT 把视频显示区定义为 VIDEO_SEG,实际上是作为一个虚拟段。在这个段的开始用 VID_AREA 确定 1 页的地址。

```
TITLE      A09DRVID (EXE) 直接视频显示, 参数为行、增序的字符
;          (A P) 和属性
.MODEL SMALL
.STACK 64
VIDEO_SEG SEGMENT AT 0B900H ;Page 1 of video area
VID_AREA DB 1000H DUP(?)
VIDEO_SEG ENDS
.286 ; -----
.CODE
A10MAIN PROC FAR
MOV AX, VIDEO_SEG ; 可寻址
MOV ES, AX ; 视频区
ASSUME ES:VIDEO_SEG
MOV AH, 0FH ; 请获取
INT 10H ; 和保存
PUSH AX ; 当前显示方式
PUSH BX ; 和显示页
MOV AX, 0003H ; 设置显示方式 03
INT 10H ; 清屏
MOV AX, 0501H ; 设置 0 页
INT 10H
CALL B10DISPLY ; 处理显示区
MOV AH, 10H ; 等待键盘
INT 16H ; 回答
MOV AH, 05H ; 恢复
POP BX ; 初始
MOV AL, BH ; 页号
INT 10H
POP AX ; 恢复显示方式
MOV AH, 00H ; (在 AL 中)
INT 10H
MOV AX, 4C00H ; 处理结束
INT 21H
A10MAIN ENDP
;
; 在视频区存放字符+属性
; 字符和属性值增量
; -----
B10DISPLY PROC NEAR
PUSHA ; 保存通用寄存器
MOV AL, 41H ; 显示字符
MOV AH, 01H ; 属性
MOV DI, B20 ; 显示区的开始
B20: MOV CX, 60 ; 每行字符数
B30: MOV ES:WORD PTR [DI], AX ; 要显示的字符
ADD DI, 2 ; 下一个属性+字符
LOOP B30 ; 重复 60 次
INC AH ; 下一个属性
INC AL ; 和字符
ADD DI, 40 ; 下一行缩进
CMP AL, 51H ; 最后字符显示?
JNE B20 ; 否, 重复
```

图 9-6 直接视频显示

```

        POPA                ; 否则,恢复寄存器
        RET                ; 并返回
B10DISPLY ENDP
        END                A10MAIN

```

图 9-6 续

该程序在第 5 行到第 20 行,第 10 列到第 69 列显示字符。第一行显示一串字符 A(41H),属性为 01H,第二行显示一串字符 B(42H),属性为 02H,以此类推,每行的字符和属性值都加 1。

程序根据在视频区定位偏移地址的规则来设立视频显示区页的起始位置:

偏移地址 = [(行×80) + 列] × 2

那么对于第 5 行,第 10 列的起始位置是 $[(5 \times 80) + 10] \times 2 = 410 \times 2 = 820$ 。显示一行之后,程序在显示区前移 40 个位置作为下一行的开始,并到达字母 Q(51H)时结束。

1 页的视频显示段定义为 VIDEO_SEG,页定义为 VID_AREA。程序设置 ES 为 VIDEO_SEG 段的段寄存器。在程序开始,保存当前显示方式和当前页,然后设置方式 03 和 01 页。

在子程序 B10DISPLY 中,AX 初始化为开始显示的字符和属性,视频显示区的起始偏移地址在 DI 中。指令 MOV ES:WORD PTR [DI], AX 传送 AL 的内容(字符)到显示区的第一个字节, AH 的内容(属性)传送到第二个字节。LOOP 程序执行这条指令在屏幕上显示字符:属性 60 次。然后增量字符:属性,DI 加 40——20 为当前行结束,20 用来确定下一行的开始(在这个屏幕上,每行 10 列)。程序又重复显示下一行的字符。

完成显示后,程序等待用户按动一个键,恢复初始的显示方式和页号。

9.8 用于方框和菜单的 ASCII 字符

在扩展的 ASCII 字符 128-225(80H-FFH)中有许多是用来显示提示符、菜单和标识语的专用字符,如图 9-7 所示。

CHARACTER	SINGLE LINE	DOUBLE LINE	MIXED LINES
Straight Lines:			
Horizontal	C4H -	CDH =	
Vertical	B3H	BAH	
Corners:			
Top left	DAH ⌈	C9H ⌈	D6H ⌈ D5H ⌈
Top right	BFH ⌋	BBH ⌋	B7H ⌋ B8H ⌋
Bottom left	C0H ⌊	C8H ⌊	D3H ⌊ D4H ⌊
Bottom right	D9H ⌋	BCH ⌋	BDH ⌋ BEH ⌋
Middle:			
Left	C3H	CCH	C7H C6H
Right	B4H	B9H	B6H B5H
Top	C2H	CBH	D2H D1H
Bottom	C1H	CAH	D0H CFH
Center Cross	C5H +	CEH +	D7H + D8H +
Blocks:			
One-quarter dots on	B0H ::	Solid shadow, upper half	DFH ■
One-half dots on	B1H ::	Solid shadow, left half	DDH
Three-quarter dots on	B2H ::	Solid shadow, right half	DEH
Solid shadow	DBH ■	Solid shadow, lower half	DCH ■

图 9-7 用于方框和菜单的 ASCII 字符

下例用 INT 10H 的功能 09H 来画一条 25 个位置长的水平实线：

```

MOV AH, 09H          ; 请求显示
MOV AL, 0C4H          ; 单实线
MOV BH, 00            ; 页号 0
MOV BL, 1EH           ; 蓝色背景, 棕色前景
MOV CX, 25            ; 重复 25 次
INT 10H               ; 调用中断服务

```

要记住, 虽然功能 09H 显示一串字符, 但它不能前移光标。

显示一个方框最简单的方法是在数据段中定义并显示整个数据区。下面的例子在一个单实线框中定义了一个菜单：

```

MENU DB 0DAH, 17 DUP(0C4H), 0BFH
      DB 0B3H, 'Add records', 0B3H
      DB 0B3H, 'Delete record', 0B3H
      DB 0B3H, 'Enter orders', 0B3H
      DB 0B3H, 'Print report', 0B3H
      DB 0B3H, 'Update accounts', 0B3H
      DB 0B3H, 'View records', 0B3H
      DB 0C0H, 17 DUP(0C4H), 0D9H, LF

```

下一章中的图 10-1 和图 10-2 举例说明了一个类似的双线框菜单, 并在方框的右边和底边加上“打点”符形成阴影。

9.9 要 点

- 视频屏幕上的水平线（光栅）含有成百个点(像素)，它们由红、绿和蓝三原色合成的荧光点组成。
- 视频显示区的数据存储在相邻的页面上。默认的页是 0 页，但是可以选择任何其他显示页。它的地址取决于当前的显示方式。
- 显示控制器的任务包括产生水平和垂直信号、跟踪视频显示区中的当前数据、监视器的水平和垂直扫描、处理光标以及选择当前页。
- 过扫描在屏幕上产生一个边界(过扫描区)。
- 文本方式的属性字节提供闪烁、反相显示、高亮度以及选择颜色的 RGB 位。
- BIOS INT 10H 提供设置显示方式、设置光标位置、滚动屏幕、选择彩色调色板、显示字符等功能。
- 在屏幕上向下显示行的程序可以调用 INT 10H 的功能 06H 在显示到达底部时上卷屏幕。
- 一个像素(或图素)由指定的位组成，其位数取决于图形适配器和分辨率。
- 图形方式 04 和 05 允许选择 4 种颜色，其中一种颜色是 16 种可用颜色之一，其他 3 种颜色来自调色板。
- 显示屏幕字符(文本或图形方式)最快的方法是直接把它们传送到视频显示区。

9.10 习 题

9-1. (a)屏幕上水平线的名称是什么? (b)线上的点的名称是什么? (c)这些点由三原色合成的荧光点组成, 这三原色是什么?

9-2. 以下方式的视频显示区的地址是什么? (a)显示方式 00H-06H, (B)单色文本方式, (C)显示方式 0DH-13H。

9-3. 指出视频控制器的 4 项任务。

9-4. (a)视频 BIOS 的两处地址是什么? (b)它们都包含些什么?

9-5. 为显示方式 03 提供页号、分辨率和颜色数。

9-6. 为下列显示提供二进制的属性: (a)黄底棕字, (b)品红底青字, (c)白底绿字, 闪烁。

9-7. 解释说明通用的属性字节是如何限制可用颜色数的。

9-8. 编写下面的程序: (a)设置 80 列彩色屏幕的文本方式; (b)设置光标大小起始于扫描行 4, 结束于扫描行 10; (c)在第 5 行, 第 20 列设置光标。

9-9. 编写下面的程序: (a)上卷屏幕 12 行并设置蓝色背景和白色前景; (b)用 1/2 打点符 (B1H)显示 40 个闪烁的“点”并设置为白色背景和红色前景。

9-10. 编写下面的程序: (a)设置活动页 3; (b)取当前的光标状态; (c)取当前显示方式。

9-11. 编写指令来显示: (a)蓝色背景黄色心形符(03H), (b)红色背景上的 10 个白色星号。(2AH)。

9-12. 编写一个程序, 用 INT 21H 的功能 0AH 从键盘上接收数据, 用 INT 10H 的功能 13H 来显示字符。清屏, 设置屏幕颜色为灰底绿字, 设置边界颜色为蓝色, 以及初始化光标为 0 行 2 列。颜色可以是变化的, 反相显示或响铃。从光标当前的位置开始接收键盘输入的数据。然后设置光标在 78 列, 以反序(反向)从右到左显示输入的数据。行增量到下一行, 往下显示数据直到 24 行, 然后开始上卷。程序应能接受任意多的输入, 当无数据输入时按下回车键结束。编写一个简短的包括主要逻辑功能的程序以及一系列的调用子程序。

9-13. 编写指令: (a)改变调色板寄存器 2 的颜色为白色, (b)改变屏幕边界的颜色为蓝色, (c)禁止闪烁属性, (d)将 256 个 DAC 寄存器都读入一个表中。

9-14. 编写指令来改变字符的大小 (a)基于文本方式的字符为 8×16 , (b)基于图形方式的字符为 8×14 , 有 25 个字符行。

9-15. 编写指令将视频 BIOS 的信息取到一个名为 VID_BIOS_TBL 的 64 字节的表中。

9-16. 编写指令设置图形方式的分辨率: (a) 320×200 , 10 种颜色, (b) 640×200 , (c) 640×480 , 16 种颜色。

9-17. 编写指令在图形方式下选择背景为红色。

9-18. 编写指令在图形方式下从第 142 行, 第 264 列读一个点。

9-19. 修改图 9-5 的程序使之提供下列条件: (a)背景为蓝色, (b)行起始于 50, 结束于 400, (c)列起始于 72, 结束于 568。

9-20. 在 9-19 题改写程序的基础上, 再将程序修改为: 每次显示一个列位置上的(代替行)图形点, 也就是说, 在屏幕上向下显示点, 然后前移到下一列, 以此类推。

目的：介绍键盘操作以及键盘输入的高级特性，包括 shift 键的状态，键盘缓冲区和扫描码。

10.1 引言

本章介绍了许多处理键盘输入的操作，其中有些操作具有专门的用途。在这些操作中，INT 21H 的功能 0AH(已包括在第 9 章)和 INT 16H(包括在本章中)将提供所需要的几乎是全部的键盘操作。

本章中的其他问题还包括键盘 shift 状态字节、扫描码以及键盘缓冲区。在 BIOS 键盘数据区中的 shift 状态字节能够使程序确定诸如 Ctrl、Shift 或 Alt 等键是否已被按下。扫描码是分配给键盘上每一个键的唯一号码，它能使系统识别按键的来源，也能使程序检验按动的键是否是扩展功能键，如 Home、PageUp 或 Arrow。键盘缓冲区提供存储器空间，以便在程序实际请求输入之前打印预先准备的内容。

本章所包括的操作如下：

INT 21H 功能

- 07H 直接键盘输入，无回显
- 08H 键盘输入，无回显
- 0AH 缓冲键盘输入
- 0BH 检验键盘状态
- 0CH 清除缓冲区，调用函数

INT 16H 功能

- 03H 设置打字速率
- 05H 键盘写入缓冲区
- 10H 读键盘字符
- 11H 确定字符是否存在
- 12H 返回键盘 Shift 状态

键盘提供 3 种基本类型的键：

1. 标准字符，由字母 A 到 Z，数字 0 到 9，以及诸如 %、\$ 和 # 等字符。
2. 扩展功能键，包括：
 - 程序功能键，如 F1 和 Shift + F1
 - 带触发开关 NumLock 键的数字键盘上的键：Home、End、Arrows、Del、Ins、PageUp 和 PageDown 以及它们在扩展键盘上的复制键。
 - Alt+字母及 Alt+程序功能键
3. 专用键 Alt、Ctrl 和 Shift，它们通常与其他键如 CapsLock、NumLock 和 ScrollLock

等联合工作以指定一个条件。BIOS 不把这些按键作为 ASCII 字符传送给程序,而是通过修改它们在 BIOS 键盘数据区中的 Shift 状态字节的当前状态,来进行不同的处理。

最初的 PC 机有 83 个键,它经历了一次缺乏远见的设计,结果使所谓的数字键盘的键执行两个动作。这样数字与 Home、End、Arrows、Del、Ins、PageUp 和 PageDown 键共用,NumLock 键在它们之间来回触发。为了解决由这种设计带来的问题,设计者为 Windows 生产了一种有 101 个键以及后来的 104 个键的增强型键盘。在增加的 18 个键中,只有 F11 和 F12 提供了新功能,其余的还是复制原来键盘上键的功能。

10.2 BIOS 键盘数据区

BIOS 数据区在存储器低地址端的段 40[0]H,该区包含有很多有用的数据项。其中包括指示控制键当前状态的两个键盘数据区。键盘数据区 1 含有两个字节,第一个字节在 40:17H,它的各位置为 1 表示下列含义:

位	作用	位	作用
7	Insert 状态变换	3	按动右 Alt 键
6	CapsLock 状态变换	2	按动右 Ctrl 键
5	NumLock 状态变换	1	按动左 Shift 键
4	Scroll Lock 状态变换	0	按动右 Shift 键

可以使用 INT 16H 的功能 02H(后面会讲到)来检验这些值。“按动”的意思是用户现在正按住键,释放键使 BIOS 清除这个位的值。

键盘数据区 1 的第二个字节在 40:18H,它的各位设置为 1 表示下列含义:

位	作用	位	作用
7	按动 Insert	3	Ctrl/NumLock(Pause) 状态变换
6	按动 CapsLock	2	按动 SysReq 键
5	按动 NumLock	1	按动左 Alt 键
4	按动 Scroll Lock	0	按动左 Ctrl 键

你可以测试这些位,例如,是否按下了 Ctrl 键或 Alt 键,或者这两个键都按下了。

在 40:80H 的键盘数据区 2 用于作键盘缓冲区,后面要讲到这个内容。键盘数据区 3 驻留在 40:96H。当位 4 为 1 指示安装了增强型键盘。键盘数据区的详细讲述可在第 24 章中找到。

练习:检查 Shift 状态。装入 DEBUG 观察按下 Ctrl、Alt 和 Shift 键时对 Shift 状态字节的影响。键入 D 40:17 来查看两个状态字节的内容。按下 CapsLock、NumLock 以及 Scroll Lock,再键入 D 40:17 来查看两个状态字节的内容。在 40:17H 的字节应当显示 70H(0111 0000),在 40:18H 的字节可能是 00H。在 40:96H 的字节(位 4)显示增强型键盘存在(1)或不存在(0)。

试着改变在 40:17H 的状态字节中的内容——敲入 E 40:17 00。如果键盘的 Lock 键的指示器是亮着的,那么它们应当是关闭的。现在再试着敲入 E 40:17 70 来打开它们。你可以尝试各种组合,虽然当按下 Ctrl 键或 Alt 键的时候,敲入一个有效的 DEBUG 命令是困难的。

键入 Q 退出 DEBUG。

键盘缓冲区

在 40:1EH 地址的 BIOS 数据区包含有一块键盘缓冲区,它允许在程序请求键盘输入之前最多键入 15 个字符。当你按动一个键时,键盘的处理器自动地产生键的扫描码(它是分配给键唯一的号码)并请求调用 BIOS INT 09H。

简单地说,INT 09H 例程从键盘取得扫描码,把它转换为 ASCII 字符,并传送到键盘缓冲区。随后,INT 16H(最低级的键盘操作)从缓冲区读字符并把它传送给程序。程序不需要请求 INT 09H,因为当按动一个键时,处理器会自动执行它。后面一节详细介绍了 INT 09H 和键盘缓冲区。

10.3 键盘输入的 INT 21H 操作

本节包括了各种处理键盘输入的 INT 21H 服务例程。所有这些操作需要 AH 中放入功能码,并且只接收一个输入字符。在下面的讨论中,术语“响应 Ctrl+Break 的请求”的意思是:如果用户按下 Ctrl+Break 或 Ctrl+C 键,系统将终止程序。

10.3.1 INT 21H 的功能 01H: 键盘输入有回显

该操作从键盘缓冲区接收一个字符,如果没有输入字符,则等待键盘输入:

```
MOV AH, 01H      ; 请求键盘输入
INT 21H
```

该操作在 AL 中返回两个状态码之一。AL=非 0 的值,说明输入了一个标准的 ASCII 字符(如字母或数字),并且在屏幕上回显。AL=0,说明用户已经按动了一个扩展功能键,如 Home 或 F1, AH 仍然保留原来的功能码。这个操作处理扩展功能键不是很巧妙,它试图在屏幕上回显扩展功能键。为了在 AL 中获得功能键的扫描码,必须立即重复一次 INT 21H 操作。本操作也响应 Ctrl+Break 的请求。

10.3.2 INT 21H 的功能 07H: 直接键盘输入无回显

该操作除了输入的字符不在屏幕上回显以及不响应 Ctrl+Break 的请求外,其工作类似于功能 01H。它一般用来键入一个不显现的密码。

10.3.3 INT 21H 的功能 08H: 键盘输入无回显

该操作除了输入的字符不在屏幕上回显之外,其工作类似于功能 01H。

10.3.4 INT 21H 的功能 0AH: 缓冲键盘输入

这个操作已经在第 9 章详细介绍了。但是由于它不能接收扩展功能键而限制了它的性能。

10.3.5 INT 21H 的功能 0BH: 检验键盘状态

如果键盘缓冲区的输入字符是可用的, 该操作在 AL 中返回 FFH; 如果没有字符是可用的, 返回 00H。注意, 这个操作不期待用户按动一个键, 它只是简单地检验缓冲区。

10.3.6 INT 21H 的功能 0CH: 清空键盘缓冲区并调用子功能

你可以利用这个操作与功能 01H、06H、07H、08H 或 0AH 联合工作。在 AL 中装入所要求的功能:

```
MOV    AH, 0CH           ; 请求键盘功能
MOV    AL, function      ; 所要求的子功能
INT     21H
```

该操作清除键盘缓冲区, 执行 AL 中的功能, 并根据功能的要求接收(或等待)一个字符。这个操作能用于不允许用户提前键入的程序。

10.4 键盘输入的 INT 16H 操作

INT 16H 是软件开发人员广泛使用的基本 BIOS 键盘操作, 该操作根据装入 AH 的功能码提供下面的一些服务例程。

10.4.1 INT 16H 的功能 03H: 设置打字重复速率

当你按下一个键超过 1/2 秒时, 键盘进入打字模式并自动地重复这个字符。为了改变速率, 可以使用如下的功能:

```
MOV    AH, 03H           ; 设置打字重复速率
MOV    AL, 05H           ; 请求子功能
MOV    BH, repeat_delay   ; 起动之前的延迟
MOV    BL, repeat_rate    ; 重复速度
INT     16H
```

BH 中的 repeat_delay 值是 0=1/4 秒, 1=1/2 秒(默认值), 2=3/4 秒, 3=1 秒。BL 中的 repeat_rate 值的范围是 0(最快)到 31(最慢)。

10.4.2 INT 16H 的功能 05H: 键盘写

该操作允许程序在键盘缓冲区中插入字符, 就好象用户按了键一样。ASCII 字符装入 CH,

它的扫描码装入 CL。该操作允许输入字符到缓冲区，直到缓冲区满。如果缓冲区已满，操作设置进位标志和 AL 为 1。

10.4.3 INT 16H 的功能 10H：读键盘字符

这个标准的键盘操作为输入字符检验键盘缓冲区。如果键盘缓冲区没有字符存在，它一直等待用户按动一个键。如果有字符存在，该操作将字符码传送到 AL，字符的扫描码传送到 AH。如果按动的字符是一个扩展功能键，如 Home 或 F1，AL 中的字符码是 00H。对于增强型的键盘，F11 和 F12 也在 AL 中返回 00H，但是其他的较新的(复制)控制键，如 Home 和 PageUp，则返回 E0H。下面是 3 种可能：

按 键	AH	AL
正常的 ASCII 字符	扫描码	ASCII 字符码
扩展功能键	扫描码	00H
扩展复制控制键	扫描码	E0H

程序可以通过测试 AL 为 00H 或 E0H 来判断是否按动了扩展功能码：

```

MOV     AH, 10H           ; 请求 BIOS 键盘输入
INT     16H               ; 调用中断服务
CMP     AL, 00H           ; 扩展功能键?
JE      exit              ; 是，则退出
CMP     AL, 0E0H          ; 扩展控制键
JE      exit              ; 是，则退出

```

因为这个操作在屏幕上不回显字符，所以程序必须请求一个屏幕显示操作，以达到显示按键的目的。

10.4.4 INT 16H 的功能 11H：确定字符是否存在

如果键盘缓冲区存在一个输入字符，该操作清除零标志位，并把字符传送给 AL，扫描码传送给 AH；输入字符仍保留在缓冲区。如果键盘缓冲区没有字符，操作设置零标志位并且不等待输入。注意，该操作提供了一个向前看的特性，因为字符保留在键盘缓冲区一直到功能 10H 来读取它。

10.4.5 INT 16H 的功能 12H：返回键盘 Shift 状态

该操作传送 BIOS 数据区 I 中地址为 40:17H 的键盘状态字节到 AL，传送 40:18H 字节到 AH。下面的例子测试 AL 以判断是否按动了左 Shift 键(位 1)或右 Shift 键(位 0)：

```

MOV     AH, 12H           ; 请求 Shift 状态
INT     16H               ; 调用中断服务
AND     AL, 00000011B     ; 按动左或右 Shift 键?
JZ      exit              ; 是，则...

```

对 AH 中的状态字节，各位为 1 的含义如下：

位	键	位	键
7	按动 SysReq	3	按动右 Alt
6	按动 Caps Lock	2	按动右 Ctrl
5	按动 Num Lock	1	按动左 Alt
4	按动 Scroll Lock	0	按动左 Ctrl

键盘练习。这里是一个测试功能 12H 的简单的 DEBUG 练习。在 DEBUG 下，输入命令 A 100 并键入下列指令：

```
MOV     AH, 12
INT     16
JMP     100
```

确定 Lock 键都是关闭的。键入命令 R、T 和 P，现在 AL 应当含有 00H。下一步打开 NumLock、CapsLock 和 ScrollLock 键，再重复这 3 条指令。这时，AL 中反映键盘状态的值应当是 70H(0111 0000)。再关闭 Lock 键并重复这些指令，但是这次按动 P 键后停止。当按<回车>时按下 CapsLock，AH 和 AL 都应当含有 40H(0100 0000)。检验 Insert 键，它的作用像一个触发键。试验用其他组合，然而有一些像 Alt+Enter 的组合不工作。

10.5 扩展功能键和扫描码

扩展功能键如 F1 或 Home 是请求一个动作而不是传输字符码。在系统设计中没有规定这些键一定要执行一个专门的动作。作为程序员，你要确定它们的动作，例如按动 Home 键就将光标设置在屏幕的左上角；按动 End 键就将光标设置在屏幕上文本的末尾。你也可以很容易地对这些键编程，使它们执行完全无关的操作。

每个键都有一个设定好的扫描码，从 Esc 的 01 开始(参见附录 F，一个完整的扫描码表)。依靠这些扫描码，程序可以确定任何按键的来源。例如，程序可以利用 INT 16H 的功能 10H 来请求输入一个字符。操作以两个方法之一来响应，这根据按动的是一个字符键还是一个扩展功能键而定。对于一个字符键，如字母 A，操作传送两项值到 AX：

AH=字母 A 的扫描码 1EH

AL=字母 A 的 ASCII 码(41H)

对有些字符键盘上有两个键，如一、+和*。例如，按动星号键，AL 中设置字符码 2AH，AH 中是两个扫描码之一，这取决于按动的是哪一个键：09H 是数字 8 上面的星号，29H 是数字键盘旁边的星号。下例测试扫描码来确定是哪个星号被按动：

```
CMP     AL, 2AH           ; 星号?
JNE     exit1             ; 不是，则退出
CMP     AH, 09H           ; 是 #8 键上的扫描码?
JE      exit2             ; 是，则退出
```

如果按动了扩展功能键，如 Del 键，则操作传送这样两个值给 AX：

AH=Del 的扫描码 53H

AL=00H(数字键盘上的 Del), 或 E0H(增强型键盘上的复制键)

执行 INT 16H 操作(以及某些 INT 21H 操作)之后, 你可以测试 AL。如果它含有 00H 或 E0H, 则是对扩展功能键的中断请求, 否则操作将传送一个字符码。下例是测试扩展功能键的指令:

```
MOV     AH, 10H           ; 请求键盘输入
INT     16H               ; 调用中断服务
CMP     AL, 00H           ; 扩展功能键?
JE      exit              ; 是, 则退出
CMP     AL, 0E0H          ; 扩展功能键?
JE      exit              ; 是, 则退出
```

在下一个例子中, 若按动了 Home 键(扫描码为 47H), 光标则设置为第 0 行, 第 0 列:

```
MOV     AH, 10H           ; 请求键盘输入
INT     16H               ; 调用中断服务
CMP     AL, 00H           ; 扩展功能键?
JE      L30               ; 是, 则跳转
CMP     AL, 0E0H          ; 扩展功能键?
JNE     exit              ; 不是, 则退出
L30:    CMP     AH, 47H    ; Home 的扫描码?
JNE     exit              ; 不是, 则退出
MOV     AH, 02H           ; 请求
MOV     BH, 00            ; 设置光标
MOV     DX, 00            ; 在 00: 00
INT     10H               ; 调用中断服务
```

功能键 F1-F10 产生的扫描码为 3BH-44H, 后面的 F11 和 F12 产生 85H 和 86H。下例测试功能键 F10:

```
CMP     AH, 44H           ; 功能键 F10?
JE      exit              ; 是, 则退出
```

1. 键盘练习。下面的 DEBUG 练习检测了键入不同的字符所产生的影响。使用命令 A 100 来键入这些指令:

```
MOV     AH, 10
INT     16H
JMP     100
```

使用 P(Proceed)命令来执行 INT 操作。输入各种常用字符与 Shift 和 Ctrl 组合, 将 AH(扫描码)和 AL(字符码)中的结果与附录 F 的表比较, 然后继续下一个练习。

2. 键入全部 ASCII 字符集。整个 ASCII 字符集由 256 个字符组成, 字符码从 0 到 255(FFH)。它们中许多是标准的可显示字符, ASCII 码从 20H(空格)到 7EH(波折号~)。由于键盘的限制, 256 个 ASCII 字符的大多数都不能在键盘上表示出来。但是你可以按下 Alt 键来输入从 01 到 255 的任何一个 ASCII 码, 并可在数字键盘上输入适当的代码作为十进制数。系统把你输入的值存储在键盘缓冲区的两个字节中: 第一个字节是产生的 ASCII 字符码, 第二个字节是 0。例如, Alt+001 传送的是 01H, Alt+255 传送的是 FFH。仍然在 DEBUG 下, 用 DEBUG 的 A

命令来检验输入不同数据的结果。并对照附录 B 的完整的 ASCII 码表。检验 AX 中的返回值

程序：从菜单中选择

下一个程序显示一个带阴影的菜单。有关解释在第 9 章，图示在图 10-1。菜单是定义在数据段中的双线框：8 行，19 个阴影符(ODBH)。用户按动上箭头或下箭头以及回车键来选择菜单中的一项。对图 10-2 程序中过程的解释如下：

- A10MAIN 调用 Q10CLEAR 清除屏幕，调用 B10MENU 显示菜单和提示符，调用 D10DISPLY 设置第一个菜单项为反相显示，调用 C10INPUT 接收键盘输入。
- B10MENU 显示全部的菜单选择项。它首先使用 INT 10H 的功能 09H 来显示阴影框，然后使用 INT 10H 的功能 13H 显示菜单(在数据段定义为 MENU，在阴影框的上方，但偏移一行和一列)。再一次使用功能 13H 来显示菜单下面的提示信息。
- C10INPUT 用 INT 16H 的功能 10H 来输入：下箭头下移菜单，上箭头上移菜单，回车键接受菜单项，Esc 退出，并忽略输入的其他键。程序使光标是可迴绕的，这样可以尝试一下移动光标将菜单的第一行移动到最后一行，反之亦然。该程序也调用 D10DISPLY 来重新设置先前的菜单为正常显示，而新的(被选择的)菜单是反相显示的。
- D10DISPLY 根据已提供的属性(正常显示或反相显示)用 INT 10H 的功能 13H 来显示当前所选择的菜单项。
- Q10CLEAR 清除整个屏幕，设置蓝色前景和棕色背景。

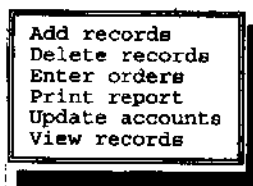


图 10-1 带阴影的菜单

```

TITLE      A10SELMU (EXE) 选择菜单项
.MODEL     SMALL
.STACK     64

;-----
.DATA
TOPROW     EQU     08           ; 菜单上部的行
BOTROW     EQU     15           ; 菜单下部的行
LEFCOL     EQU     26           ; 菜单左边的列
ATTRIB     DB      ?           ; 屏幕属性
ROW        DB      00           ; 屏幕行
SHADOW     DB      19 DUP(ODBH) ; 阴影符
MENU        DB      0C9H, 17 DUP(0CDH), 0BBH
            DB      0BAH, ' Add records      ', 0BAH
            DB      0BAH, ' Delete records   ', 0BAH
            DB      0BAH, ' Enter orders    ', 0BAH
            DB      0BAH, ' Print report    ', 0BAH
            DB      0BAH, ' Update accounts ', 0BAH
            DB      0BAH, ' View records    ', 0BAH
            DB      0C8H, 17 DUP(0CDH), 0BCH
PROMPT     DB      'To select an item, use <Up/Down Arrow>'
            DB      ' and press <Enter>..'
            DB      13, 10, 'Press <Esc> to exit..'

```

图 10-2 从菜单中选择一项

```

.386 ; -----
A10MAIN .CODE
        PROC FAR
        MOV AX,@data ; 初始化段寄存器
        MOV DS,AX
        MOV ES,AX
        CALL Q10CLEAR ; 清屏
        MOV ROW,BOTROW+4 ; 设置行

A20:
        CALL B10MENU ; 显示菜单
        MOV ROW,TOPROW+1 ; 设置上部菜单项的行
        MOV ATTRIB,16H ; 设置反相显示
        CALL D10DISPLY ; 当前菜单行高亮度
        CALL C10INPUT ; 菜单选择
        CMP AL,1BH ; 按下Escape键?
        JNE A20 ; 否,继续
        MOV AX,0600H ; 是,结束
        CALL Q10CLEAR ; 清屏
        MOV AX,4C00H ; 处理结束
        INT 21H
A10MAIN ENDP
;
; ----- 显示阴影框、菜单以及提示符 -----
;
B10MENU PROC NEAR
        PUSHAX ; 保存通用寄存器
        MOV AX,1301H ; 请求显示阴影框
        MOV BX,0060H ; 页和属性
        LEA BP,SHADOW ; 阴影符
        MOV CX,19 ; 19个字符
        MOV DH,TOPROW+1 ; 阴影上边的行
        MOV DL,LEFCOL+1 ; 阴影左边的列

B20:
        INT 10H
        INC DH ; 下一行

        CMP DH,BOTROW+2 ; 显示完所有行?
        JNE B20 ; 否,重复
        MOV ATTRIB,71H ; 白底蓝字
        MOV AX,1300H ; 请求显示菜单
        MOVZX BX,ATTRIB ; 页和属性
        LEA BP,MENU ; 菜单行
        MOV CX,19 ; 行的长度
        MOV DH,TOPROW ; 行
        MOV DL,LEFCOL ; 列

B30:
        INT 10H
        ADD BP,19 ; 菜单下一行
        INC DH ; 下一行
        CMP DH,BOTROW+1 ; 所有行显示完?
        JNE B30 ; 否,重复

        MOV AX,1301H ; 请求显示提示符
        MOVZX BX,ATTRIB ; 页和属性
        LEA BP,PROMPT ; 提示符的行
        MOV CX,79 ; 行的长度
        MOV DH,BOTROW+4 ; 屏幕行,
        MOV DL,00 ; 列
        INT 10H
        POPA ; 恢复寄存器
        RET
B10MENU ENDP
;
; ----- 接收键盘请求,箭头和回车用来选择菜单行,
; Esc退出: -----
;
C10INPUT PROC NEAR
        PUSHAX ; 保存通用寄存器
C20:
        MOV AH,10H ; 从键盘请求一个
        INT 16H ; 字符
        CMP AH,50H ; 下箭头?
        JE C30
        CMP AH,48H ; 上箭头?
        JE C40
        CMP AL,0DH ; 回车键?

```

图 10-2 续


```

                JE      C90
                CMP     AL,1BH                ;Escape 键?
                JE      C90
                JMP     C20
C30:            MOV     ATTRIB,71H            ; 都不是,重新输入
                CALL    D10DISPLY            ; 白底蓝字
                INC     ROW                  ; 设置原来的行为正常显示
                CMP     ROW,BOTROW-1         ; 加1到下一行
                JBE     C50                  ; 超过了底边的行?
                MOV     ROW,TOPROW+1         ; 否,继续
                JMP     C50                  ; 是,重新设置
C40:            MOV     ATTRIB,71H            ; 白底蓝字
                CALL    D10DISPLY            ; 设置原来的行为正常显示
                DEC     ROW
                CMP     ROW,TOPROW+1         ; 在上边行下面?
                JAE     C50                  ; 否,继续
                MOV     ROW,BOTROW-1         ; 是,重新设置

C50:            MOV     ATTRIB,17H            ; 蓝底白字
                CALL    D10DISPLY            ; 设置新行为反相显示
                JMP     C20
C90:            POFA
                RET
C10INPUT       ENDP
;
;                      设置菜单行为高差度(如被选)
;                      或正常差度(如未被选):
;
D10DISPLY      PROC    NEAR
                PUSH    AX
                MOVZX   AX,ROW                ; 保存通用寄存器
                SUB     AX,TOPROW            ; Row 通知哪行要设置
                IMUL    AX,19                ; 乘以行的长度
                LEA     SI,MENU+1            ; 选择的菜单行
                ADD     SI,AX

                MOV     AX,1300H            ; 请求显示
                MOVZX   BX,ATTRIB            ; 页和属性
                MOV     BP,SI                ; 字符串
                MOV     CX,17                ; 串长度
                MOV     DH,ROW              ; 行
                MOV     DL,LEFCOL+1         ; 列
                INT     10H
                POFA
                RET
D10DISPLY      ENDP
;
;                      清屏:
;
Q10CLEAR       PROC    NEAR
                PUSH    AX
                MOV     AX,0600H            ; 保存通用寄存器
                MOV     BH,61H              ; 棕底蓝字
                MOV     CX,0000            ; 全屏
                MOV     DX,184FH
                INT     10H
                POFA
                RET
Q10CLEAR       ENDP
                END     A10MAIN

```

图 10-2 续

这个程序以一个较为简单的方式分析了菜单选择,对应每个选择项整个程序会执行相应的例程。为了更好地理解这个程序,汇编、测试并扩充它。

10.6 BIOS INT 09H 和键盘缓冲区

键盘有一个 8 位的 Intel 8048 处理器,用来判断按下键和释放键。根据这些被按动的键,键盘将其代码送到系统板上另一个 8 位的处理器 Intel 8042。当按动一个键时,键盘的处理器

产生键的扫描码并请求 INT 09H。这个中断(在中断向量表的 36 地址)是指向 ROM BIOS 中的一个中断处理例程。这个中断程序从端口 96(60H)发出一个输入请求: IN AL, 60H。这个 BIOS 例程读取扫描码, 并为了获得相关的 ASCII 字符(如果有的话)与输入的一个扫描码表进行比较。该例程将扫描码和相关的 ASCII 字符组合在一起并传送到键盘缓冲区的两个字节中。图 10-3 说明了这个过程。

值得注意的是, INT 09H 处理键盘数据区中的 40:17H、40:18H 和 40:96H 单元的 Shift、Alt 和 Ctrl 的状态字节。虽然按动这些键产生 INT 09H 调用, 该中断例程设置状态字节的相应位, 但不传送任何字符到键盘缓冲区。INT 09H 也不处理未定义的按键组合, 例如 Ctrl+/-。

当按动一个键时, 键盘处理器自动地产生它的扫描码, 并请求 INT 09H 中断。当你在 1/2 秒内放开这个键, 它产生第二个扫描码(第一个扫描码的值加上 1000 0000B, 即置最高位为 1), 并发出另一个 INT 09H 中断。这第二个扫描码告诉中断例程你已经释放了这个键。如果你按一个键超过 1/2 秒, 则键盘处理为打字, 也就是说, 它将自动地重复这个键的操作。

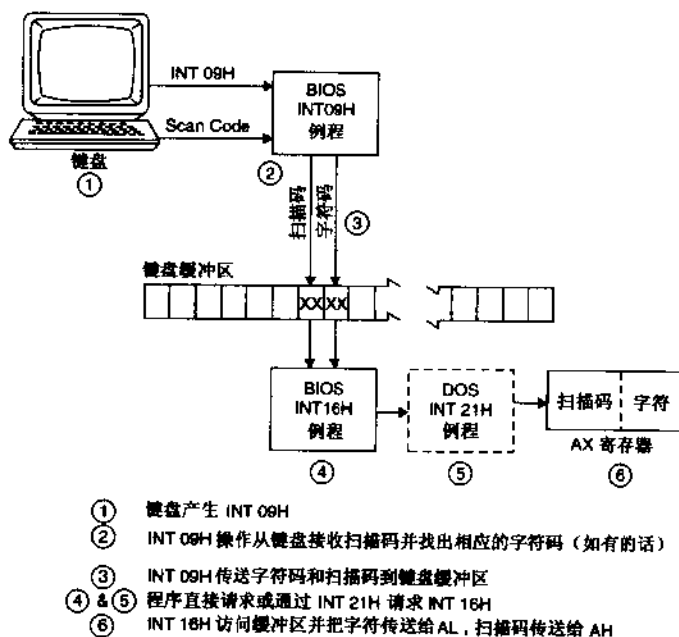


图 10-3 键盘缓冲区

键盘缓冲区需要一个地址(缓冲区的头地址)来告诉 INT 16H 例程从什么地方读下一个字符, 还需要另一个地址(缓冲区的末尾地址)告诉 INT 09H 在什么地方存储下一个字符。这两个地址分别是键盘数据区 1 的偏移地址 41AH 和 41CH。下面说明缓冲区的内容:

地址 解释

- 41AH 当前缓冲区的头地址, INT 16H 从下一个字节获得字符并送给程序的 AX 中。
 41CH 当前缓冲区的末尾地址, INT 09H 将键盘输入的字符存入下一个字节。
 41EH 键盘缓冲区的起始地址。该缓冲区包含有 16 个字(32 字节), 虽然它还可以更长, 缓冲区保存输入的键盘字符及其相应的扫描码。随后 INT 16H 将读取每

个字符和它的扫描码，然后把它们传送给程序。每个字符和它的扫描码需要 2 个字节。

当用户键入一个字符，INT 09H 前移缓冲区的尾指针。当 INT 16H 读一个字符时前移头指针。这样头指针不断地紧跟着尾指针，处理就是循环的。当缓冲区为空时(INT 16H 已经读取了全部的存储字符)，头指针和尾指针指向同一个地址。

在下面的例子中，头指针和尾指针都初始化为 41EH。然后键入字符 ‘abc<Enter>’，INT 09H 将它们存储在如下地址：

- ‘a’ 在缓冲区的 41EH，它的扫描码 1EH 在 41FH；
- ‘b’ 在缓冲区的 420H，它的扫描码 30H 在 421H；
- ‘c’ 在缓冲区的 422H，它的扫描码 2EH 在 423H；
- Enter 在 424H，它的扫描码 E0H 在 425H；

这时，INT 09H 前移尾指针到 426H：

字符	a	1EH	b	30H	c	2EH	<Ent>	E0H	...
地址	41E	41F	420	421	422	423	424	425	426

程序第一次发出 INT 16H，该操作读取 “a” 及其扫描码，并前移头指针到 420H。一旦程序发出了 4 次 INT 16H，它就读出了全部字符并前移头指针到 426H。因为尾指针和头指针已经指向同一个地址，所以缓冲区为空。

当键入 15 个字符，缓冲区就满了，尾指针直接移到了头指针之后。为了看到这种情况，假设你现在键入了 ‘fghijklmnopqrs<Enter>’。INT 09H 从尾指针所在的 426H 字节开始存储字符，循环存储 ‘<Enter>’ 在 422H。现在尾指针前移到 424H，直接在头指针的 426H 之后：

r	s	<Ent>	*	f	g	h	i	j	k	l	m	n	o	p	q
41E	420	422	424	426	428	42A	42C	42E	430	432	434	436	438	43A	43C

这时，INT 09H 不再接收任何键入的字符，虽然缓冲区能保存 16 个字符，但它确实最多只接收了 15 个字符。(你能说出这是为什么吗？如果 INT 09H 又接受了一个字符，尾指针将移到和头指针相同的地址，INT 16H 将会错误地认定缓冲区是空的。)

Shift、Ctrl 和 Alt 键

INT 09H 也能处理 BIOS 数据区中 40:17H、40:18H 和 40:96H 的键盘状态字节。当按动 Shift、Ctrl 或 Alt 键时，BIOS 例程设置相应的位为 1，当释放键时，它又清除这些位为 0。要注意的是仅按动一个控制键是不满足 INT 16H 的要求的，必须按下控制键的同时又按下另一个键，这才会引起一次有效的键盘输入，如 Shift+A，Ctrl+F1 等(见附录 F)。键盘状态字节反映了控制键的作用。程序可以依靠 INT 16H 的功能 12H 来判断是否按动了某一个控制键。

图 10-4 的程序说明了如何存取和测试在 40:17H 的状态字节。作为一个惯例，最好通过编程练习，用 BIOS 中断来存取 BIOS 数据区，但是这个程序展示了许多汇编语言的技巧。程序包含有下面几个过程：

- A10MAIN 等待用户按键。按动回车键通知程序结束, 如果不是回车键, 程序发出 INT 16H 的功能 12H 来取得键盘状态。如果与一个有效的键一起按动了 Shift、Ctrl 或 Alt 键中的一个键, 则程序调用 B10DISPLY。
- B10DISPLY 显示按动的控制键的相应信息。

```

TITLE      A10KBSTA (EXE)      测试 Alt、Shift、Ctrl 的状态
.MODEL     SMALL
.STACK    64
.DATA
BIODATA    SEGMENT AT 40H      ; 定位 BIOS 数据区
ORG        17H                ; 和
KBSTATE    DB ?               ; 状态字节
BIODATA    ENDS
; -----
CR          EQU    0DH         ; 回车
LF          EQU    0AH         ; 换行
ALTKEY      DB      'Alt key pressed ', CR, LF
CTRLKEY     DB      'Ctrl key pressed ', CR, LF
SHIFTKEY    DB      'Shift key pressed', CR, LF
.286 ; -----
.CODE
A10MAIN     PROC FAR
MOV         AX, BIODATA        ; 初始化 BIODATA
MOV         ES, AX             ; 的段址在 ES 中
A20:        MOV         AH, 10H ; 请求键盘输入
INT         16H
CMP         AL, 0DH            ; 用户请求结束?
JE          A90                ; 是, 退出
MOV         BL, ES:KBSTATE     ; 取键盘状态字节
TEST        BL, 00000011B      ; 按下 Shift+char?
JZ          A30                ; 否, 跳转
LEA         BP, SHIFTKEY       ; 请求显示
CALL        D10DISPLY          ; Shift 信息
A30:        TEST        BL, 00000100B ; 按下 Ctrl+char?
JZ          A40                ; 否, 跳转
LEA         BP, CTRLKEY        ; 请求显示
CALL        D10DISPLY          ; Ctrl 信息
A40:        TEST        BL, 000001000B ; 按下 Alt+char?
JZ          A20                ; 否, 跳转
LEA         BP, ALTKEY         ; 请求显示
CALL        D10DISPLY          ; Alt 信息
JMP         A20                ; 重复
A90:        MOV         AX, 4C00H ; 处理结束
INT         21H
A10MAIN     ENDP
;
; 如按下键显示 Alt、Ctrl 和 Shift 信息
D10DISPLY   PROC NEAR          ; BP 已设置输入数据
PUSHA                          ; 保存寄存器
PUSH        ES
MOV         AX, @data           ; 为 ES:BP
MOV         ES, AX             ; 设置数据地址
MOV         AX, 1301H           ; 请求显示
MOV         BX, 0016H          ; 页和属性
MOV         CX, 17              ; 串长度
MOV         DX, 1008H          ; 行和列
INT         10H
POP         ES
POPA                          ; 恢复寄存器
RET
D10DISPLY   ENDP
END         A10MAIN

```

图 10-4 检验键盘状态字节

你可以修改这个程序, 最好是测试位于 40:18H 的键盘状态字节。

10.7 要 点

- 在 BIOS 键盘数据区的 40:17H 和 40:18H 的 Shift 状态字节指出如 Ctrl、Alt、Shift、CapsLock、NumLock 和 ScrollLock 等键的当前状态。
- INT 21H 键盘操作提供了在屏幕上回显或不回显、认可或忽略 Ctrl+Break 以及接收扫描码等多种服务例程。
- INT 16H 的功能 10H 提供了从键盘缓冲区接收字符的基本 BIOS 键盘操作。对字符键，操作传送字符码给 AL，键的扫描码给 AH。对于一个扩展功能键，该操作传送 00H 或 E0H 给 AL，键的扫描码给 AH。
- 扫描码是分配给每一个键的唯一的号码，它能使 BIOS 识别按键的来源，也能使程序对扩展功能键如 Home、PageUp 和 Arrow 等进行检验。
- BIOS 键盘数据区的 40:1EH 包含有键盘缓冲区，它允许程序请求输入之前键入最多 15 个字符。
- 按下一个键使得键盘处理器产生键的扫描码，并请求 INT 09H。释放一个键使其产生第二个扫描码(第一个扫描码加上 10000000B，即将其最左边 1 位置为 1)，以通知 INT 09H 键已经释放。
- BIOS INT 09H 从键盘取得一个扫描码。该操作作用这个扫描码产生一个对应的 ASCII 字符码，并传送到键盘缓冲区。BIOS 也可以设置 Ctrl、Alt 和 Shift 键的状态。

10.8 习 题

10-1. (a)键盘 Shift 状态的第一个字节在 BIOS 数据区中的地址是什么？(b)值 00000010 的意思是什么？(c)值 00001100 的意思是什么？

10-2. 描述下列 INT 21H 键盘输入功能的特性：(a)01H，(b)07H，(c)08H，(d)0AH。

10-3. 解释 INT 16H 的功能 11H 与功能 10H 的不同。

10-4. 提供下列按键的扫描码：(a)End，(b)PageUp，(c)Arrow，(d)功能键 F6。

10-5. 利用 DEBUG 来测试输入键的效果。为了输入汇编语句，敲入 A 100 并键入下列指令：

```
MOV     AH, 10
INT     16
JMP     100
```

用 U 100, 104 来反汇编程序，用 P 命令来使 DEBUG 执行 INT 指令。执行停止，等待输入。按下一个键并检测 AH 和 AL。继续输入各种不同的键。按下 Q 退出 DEBUG。

10-6. 编写指令用 INT 16H 的功能 10H 来接收按键，如果是 PageDown，设置光标到第 0 列，第 24 行。

10-7. 修改图 10-2 的程序以提供下列功能：(a)将全阴影修改为 3/4 打点符(B2H)。(b)初始化清屏后，显示一个提示符，请用户按动 F1 显示菜单屏幕。(c)当按动了 F1，显示菜单。

(d)允许用户通过按下每项的第一个字符(大写或小写)来选择菜单项。(e)执行对一项的请求,显示反映这个特殊选择的信息,如“Procedure to Delete Records.”。(f)在菜单的最后一行加上一项“Exit from program”,允许用户结束处理过程。必须修改过程BIOMENU来处理另一行的显示。

10-8. 在什么情况下会产生INT 09H的操作?

10-9. 用简单的术语来解释INT 09H处理Alt和Ctrl键与处理标准键盘的键的不同之处。

10-10. (a)键盘缓冲区在BIOS存储器中的地址是什么?(b)缓冲区的大小是多少字节?(c)缓冲区能够包含多少键盘字符?

10-11. 解释在键盘缓冲区发生下列情况所带来的影响:(a)尾指针的地址紧跟着头指针。(b)头指针的地址和尾指针的地址相同。

10-12. 按下列要求修改图10-4的程序:(a)传送键盘Shift状态的第二个字节的内容到BH。(b)测试按动左Alt和左Ctrl键,并显示适当的信息。

第四部分

数据操作

目的：说明用于处理串数据的专用指令

11.1 引言

到本章为止，已提供的指令所处理的数据都是只按一个字节、字或双字定义的，但是，我们经常需要传送和比较超过这些长度的数据字段。例如，你可能要比较货名或名字，目的是把它们排序成升序序列。这种类型的项称为**串数据**，并且可以是字符或数字格式。汇编语言为处理串数据提供以下串指令：

- MOVS** 把一个字节、字或双字从存储器的一个单元传送到另一个单元。
- LODS** 从存储器取一个字节送到 AL，取一个字送到 AX，或取一个双字送到 EAX。
- STOS** 把 AL，AX，或 EAX 的内容存入存储器。
- CMPS** 比较在存储器中的字节、字或双字的项。
- SCAS** 把 AL，AX，或 EAX 的内容和存储器中的一个项的内容进行比较。

一种相关联的指令前缀 **REP** 使串指令重复执行，使它可以按指定的次数处理任意数量的字节、字或双字。另外两条串指令 **INS** 和 **OUTS** 将在第 24 章介绍。

11.2 串操作的特点

每条串指令有字节、字和双字形式来做重复处理，并且成对使用 **ES: DI** 或 **DS: SI** 寄存器。因此，你可以对具有奇数字节的串选择字节操作，而对于具有偶数字节的串选择字操作。使用字和双字操作可以提供更快的处理速度。

串指令希望 **DI** 和 **SI** 寄存器包含有效的偏移地址，用它去引用存储器中的字节。**SI** 通常和 **DS**(数据段)联系在一起成为 **DS:SI**，而 **DI** 总是和 **ES**(附加段)联系在一起成为 **ES:DI**。由于这个原因，**MOVS**、**STOS**、**CMPS** 和 **SCAS** 通常要求 **EXE** 程序初始化 **ES**，但这不是必须的，它可以用 **DS** 中的同样的地址：

```
MOV  AX, @data           ; 得到数据段的地址,
MOV  DS, AX              ;  把它存到 DS 和
MOV  ES, AX              ;  ES 中
```

为零或 CX 减到零时, 停止重复。

- **REPNE 或 REPNZ**: 当 ZF 指出不相等/不为零时, 重复该操作; 当 ZF 指出相等/为零或当 CX 减到零时, 停止重复。

下面各节要详细分析各种串操作。

11.3 MOVS: 串传送指令

MOVS, **MOVSB**, **MOVSW**, 以及 **MOVSD** 和 **REP** 前缀组合在一起, 并且长度在 CX 中, 可以传送指定数量的字符。段: 偏移寄存器对于接收串, 是 ES: DI; 对于发送串, 是 DS: SI。因此, 在 EXE 程序的起点, 一定要把 ES 和 DS 一起初始化, 并且在执行 **MOVS** 以前, 也要初始化 DI 和 SI。根据方向标志, **MOVS** 使 DI 和 SI 对于字节是加 1 或减 1, 对于字是加 2 或减 2, 对于双字则是加 4 或减 4。下面的例子说明 12 个字的传送:

```
MOV CX, 12           ; 字数
LEA DI, RECV_STR     ; RECV_STR 的地址 (ES: DI)
LEA SI, SEND_STR     ; SEND_STR 的地址 (DS: SI)
REP MOVSW            ; 传送 12 个字
```

等效于 **REP MOVSW** 操作的指令是:

```
      JCXZ L40        ; 若 CX 最初为零则绕过
L30:  MOV AX, [SI]     ; 从 SEND_STR 取字
      MOV [DI], AX    ; 把字存入 RECV_STR
      ADD DI, 2       ; 为下一个字增量
      ADD SI, 2       ;
      LOOP L30        ; CX 减量并重复
L40:  ...
```

在较早的时候, 图 6-2 已说明了 9 字节字段的传送, 为此目的, 该程序也可以使用 **MOVSB**。在图 11-2 的部分程序中, ES 是被初始化的, 这是由 **MOVS** 指令所要求的。该程序使用 **MOVSB** 传送 12 个字节的字段 **STRING1** 到 **STRING2**, 每次一个字节。第一条指令 **CLD** 清除方向标志, 使 **MOVSB** 从左到右处理数据。在开始执行时, 方向标志通常是 0, 这里 **CLD** 的编码是作为一种预防措施。

MOV 指令用 12(**STRING1** 和 **STRING2** 的长度)初始化 CX, 两条 **LEA** 指令是把 **STRING1** 和 **STRING2** 的偏移地址分别装入 SI 和 DI。现在 **REP MOVSB** 完成以下操作:

- 把 **STRING1** 的最左边字节(按 DS: SI 寻址)传送到 **STRING2** 的最左边字节(按 ES: DI 寻址);
- 为下一个右边的字节将 DI 和 SI 加 1;
- CX 减 1;
- 重复这一操作 12 次, 直到 CX 变成 0 为止。

由于方向标志是 0 并且 **MOVSB** 使 DI 与 SI 加 1, 所以每次迭代处理右边一个字节, 如 **STRING1+1** 到 **STRING2+1**, 等等。在执行结束时, CX 的内容是 0, DI 的内容是 **STRING2+12** 的地址, 而 SI 的内容是 **STRING1+12** 的地址——二者都包含超过名字末尾的一个字节(即指

向串所定义的数据项后的第一个字节)。

为了从右到左地处理,要把方向标志置成 1。然后 MOVSB 使 DI 和 SI 减 1,但为了正确地传送内容,你必须用 STRING1+11 初始化 SI,用 STRING2+11 初始化 DI。

接下去的程序使用 MOVSW 从 STRING2 到 STRING3 传送 6 个字。在执行结束时, CX 是 0, DI 是 STRING3+12 的地址, SI 则是 STRING2+12 的地址。

因为 MOVSW 使 DI 和 SI 加 2,该操作只要求重复 6 次。为了从右到左处理过程,要置方向标志,并用 STRING1+10 初始化 SI,用 STRING2+10 初始化 DI。

```

STRING1 DB 'Interstellar' ;数据项
STRING2 DB 12 DUP(' ')
STRING3 DB 12 DUP(' ')

...
MOV AX,@data ;初始化
MOV DS,AX ;段寄存器
MOV ES,AX ;
;使用 MOVSB:
;从左向右
;移 12 个字节,
LEA DI,STRING2 ;STRING1 到 STRING2
LEA SI,STRING1
REP MOVSB

;使用 MOVSW:
;从左向右
;移 6 个字,
LEA DI,STRING3 ;STRING2 到 STRING3
LEA SI,STRING2
REP MOVSW
...
```

图 11-2 使用 MOV 串操作

11.4 LODS: 从串取指令

LODS 简单地从存储器取一个字节装入 AL,取一个字装入 AX,取一个双字装入 EAX。尽管可以不考虑 SI,但存储器地址还是受 DS:SI 寄存器支配的。根据方向标志,该操作使 SI 加 1 或减 1(对于字节),加 2 或减 2(对于字),加 4 或减 4(对于双字)。

因为一个 LODS 操作填满了寄存器,所以没有什么实际理由要在它前面使用 REP 前缀。对于大多数用途,简单的 MOV 指令已经可以满足要求。但是 MOV 产生 3 个字节的机器码;而 LODS 却只产生一个字节的机器码,尽管你必须初始化 SI。你可以使用 LODS,每次一个字节、一个字或一个双字步进地通过一个串,对一个特殊的字符进行逐次地检查。

等效于 LODSB 的指令是:

```

MOV AL,[SI] ;把字节传送到 AL
INC SI ;为下一个字节增 1
```

下面的例子定义一个名为 STRING1 的 12 字节的字段和名为 STRING2 的另一个 12 字节的字段,STRING1 包含的值为“InterStellar”。目的是以相反的顺序把 STRING1 的字节传送到 STRING2,使得 STRING2 含有“ralletsretnI”。LODSB 每次从 STRING1 取一个字节送入 AL,而 MOV [DI],AL 则把该字节传送到 STRING2,方向是从右到左。

```

STRING1    DB  'interstellar'      ; 数据项
STRING2    DB  12 DUP(20H)
...
CLD                          ; 从左到右
MOV  CX, 12
LEA  SI, STRING1             ; STRING1 的地址 (DI: SI)
LEA  DI, STRING2+11          ; STRING2+11 的地址 (ES: DI)
L20:  LODS                    ; 取得字符到 AL 中,
      MOV  [DI], AL           ; 存入 STRING2,
      DEC  DI                 ; 从右到左
      LOOP L20                ; 是 12 个字符吗?
      ...

```

11.5 STOS: 存入串指令

STOS 把 AL、AX 或 EAX 的内容存入存储器中的一个字节、一个字或一个双字。存储器地址总是受 ES: DI 支配的。根据方向标志, STOS 使 DI 加 1 或减 1(对于字节), 加 2 或减 2(对于字), 加 4 或减 4(对于双字)。

带 REP 前缀的 STOS 的实际用法是把数据区初始化为任意指定值, 比如把一个区域清除成为空白。把字节、字或双字的数量设置在 CX 中, 等效于 REP STOSB 的指令是:

```

      JCXZ  L30                ; 若 CX 为零则转移
L20:  MOV  [DI], AL             ; AL 存入存储器
      INC/DEC DI                ; 增量或减量(设置标志)
      LOOP L20                 ; CX 减量并重复
L30:  ...                      ; 操作完成

```

在以下的例子中, STOSW 指令重复 6 次把含有 2020H(空白)的字存储到 STRING1 中。该操作把 AL 存入第一个字节, 而把 AH 存入下一个字节(即相反顺序)。最后, 所有 STRING1 都是空白, CX 是 00, 而 DI 则是 STRING1+12 的地址。

```

CLD                          ; 从左到右
MOV  AX, 2020H                ; 传送
MOV  CX, 06                    ; 6 个空白字
LEA  DI, STRING1               ; 到 STRING1 (ES: DI)
REP  STOSW

```

11.6 程序: 使用 LODS 和 STOS 编辑数据

图 11-3 的程序说明 LODS 和 STOS 两条指令的用法, 其目的是允许用户编辑字符串。为了减少所需时间与复杂性, 所以这是一个梗概的编辑程序。基本上, 程序是显示一个 30 个字符的字符串。过程实现如下:

- A10MAIN 初始化可寻址性(addressability), 调用 Q30DISPLY 显示串, 并且调用

B10KEYBRD 请求键盘字符。当用户按<Esc>键时, 程序结束。

```

TITLE      ALL EDIT (EXE)   Editing Features
.MODEL     SMALL
.STACK     64
.DATA

INDENT     EQU      24
LEFTLIM    EQU      00
RIGHTLIM   EQU      29
NOCHARS    EQU      30
COL        DB       00
ROW        DB       10
DATASTR    DB       'abcdefghijklmnopqrstuvwxyz'
           DB       'pqrstuvwxyzABCD', 20H

.386
;-----
.CODE
A10MAIN    PROC        FAR
MOV        AX,@data
MOV        DS,AX
MOV        ES,AX
CALL       Q10CLEAR
CALL       Q20CURSOR
CALL       Q30DISPLY

A30:
CALL       Q20CURSOR
CALL       B10KEYBRD
CMP        AH,01H
JNE        A30
MOV        AX,0600H
CALL       Q10CLEAR
MOV        AX,4C00H
INT        21H

A10MAIN    ENDP

;
; 取得键盘字符并确定要做的动作:
;-----
B10KEYBRD  PROC        NEAR
MOV        AH,10H
INT        16H
CMP        AL,00H
JE         B20
CMP        AL,0E0H
JE         B20
CALL       H10CHARS
JMP        B90

B20:
CMP        AH,4DH
JNE        B30
CALL       C10RTARRW
JMP        B90

B30:
CMP        AH,4BH
JNE        B40
CALL       D10LFARRW
JMP        B90

B40:
CMP        AH,53H
JNE        B50
CALL       E10DELETE
JMP        B90

B50:
CMP        AH,47H
JNE        B60
CALL       F10HOME
JMP        B90

B60:
CMP        AH,4FH
JNE        B90
CALL       G10END

B90:
RET
B10KEYBRD  ENDP

```

图 11-3 简单的编辑指令

```

;
;          右箭头。如在最右边，则置光标至最左边，否则列加1：
;
C10RTARRW PROC    NEAR
CMP        COL, RIGHTLIM    ; 在最右边?
JAE        C20              ; 是，
INC        COL              ; 否，列加1
JMP        C90              ; 退出
C20:       CALL    F10HOME    ; 光标到最左边
C90:       RET
C10RTARRW ENDP

;
;          左箭头。如在最左边，则置光标至最右边，否则列减1：
;
D10LFARRW PROC    NEAR
CMP        COL, LEFTLIM     ; 在最左边?
JBE        D20              ; 是，
DEC        COL              ; 否，列减1
JMP        D90              ; 退出
D20:       CALL    G10END     ; 光标到最右边
D90:       RET
D10LFARRW ENDP

;
;          删除键。用其右边字符取代当前字符，最右边字符向左移位：
;
E10DELETE PROC    NEAR
MOVZX     BX, COL           ; 使用 BX, DI, SI
PUSH      BX               ; 取列
LEA       DI, [DATASTR+BX] ; 保存列
LEA       SI, [DATASTR+BX+1] ; 初始化当前列
; 和相邻列
E20:      LODSB             ; 把相邻字符
STOSB     ; 存入当前列
CALL     Q40DISCHR          ; 显示字符
INC      COL               ; 下一列增量
CALL     Q20CURSOR          ; 设置光标
CMP      COL, RIGHTLIM     ; 在最右边吗?
JBE      E20               ; 否，重复
POP      BX                ; 取保存的原始
MOV      COL, BL           ; 列
RET
E10DELETE ENDP

;          Home 键。置光标到左列：
;
F10HOME PROC    NEAR
MOV      COL, LEFTLIM      ; 设置光标
CALL     Q20CURSOR          ; 到最左边
RET
F10HOME ENDP

;          End 键。置光标到右列：
;
G10END PROC    NEAR
MOV      COL, RIGHTLIM     ; 设置光标
CALL     Q20CURSOR          ; 到最右边
RET
G10END ENDP

;          所有其他键。绕过低于
;          20H 和高于 7EH 的字符，其他的插入到光标处：
;
H10CHARS PROC    NEAR
; 使用 BX, DI
CMP      AL, 20H            ; 是低于 20H 的 ASCII 字符吗?
JB       H90                ; 是，绕过
CMP      AL, 7EH            ; 高于 7EH 吗?
JA       H90                ; 是，绕过
MOVZX    BX, COL            ; 用 COL 作为变址

```

图 11-3 续

```

                                MOVZX  BX, COL           ; 用COL 作为变址
                                LEA     DI, DATASTR      ; 传送字符串到
                                MOV     {DI+BX}, AL       ; 数据串
                                CALL    Q40DISCHR        ; 显示字符
                                CMP     COL, RIGHTLIM     ; 在最右边?
                                JAE     H90               ; 是, 退出
                                INC     COL               ; 否, 列加1
H90:                             RET
H10CHARS ENDP

;
; 清除屏幕并设置属性:
;
Q10CLEAR PROC NEAR
MOV     AX, 0600H           ; 请求上卷
MOV     BH, 61H            ; 褐底蓝字
MOV     CX, 0000           ; 全屏幕
MOV     DX, 184FH
INT     10H
RET
Q10CLEAR ENDP

;
; 设置光标 行: 列:
;
Q20CURSOR PROC NEAR
MOV     AH, 02H            ; 请求设置光标
MOV     BH, 00             ; 0 页
MOV     DH, ROW            ; 行
MOV     DL, COL            ; 列
ADD     DL, INDENT          ; 在屏幕上缩进排
INT     10H
RET
Q20CURSOR ENDP

;
; 显示完整数据串:
;
Q30DISPLY PROC NEAR
MOV     AX, 1301H          ; 请求显示
MOV     BX, 0016H          ; 页, 属性
LEA     BP, DATASTR       ; 数据行
MOV     CX, NOCHARS+1      ; 行的长度
MOV     DH, ROW            ;
MOV     DL, COL            ;
ADD     DL, INDENT          ; 屏幕上缩进排
INT     10H
RET
Q30DISPLY ENDP

;
; 显示单个字符:
;
Q40DISCHR PROC NEAR
MOV     AH, 0AH            ; 字符在 AL 中
MOV     BH, 00             ; BIOS入口
MOV     CX, 01             ; 页
MOV     CX, 01             ; 一个字符
INT     10H
RET
Q40DISCHR ENDP
END A10MAIN

```

图 11-3 续

- B10KEYBRD 接受键盘字符: 如果按<RightArrow>键, 则调用 C10RTARRW; 如果按<LeftArrow>键, 则调用 D10LFARRW; 如果按键, 则调用 E10DELETE; 如果按<Home>键, 则调用 F10HOME; 如果按<End>键, 则调用 G10END; 否则, 对于所有其他字符调用 H10CHARS。
- C10RTARRW 向右移动光标; 如果光标已经在最右边了, 那就调用 F10HOME。
- D10LFARRW 向左移动光标; 如果光标已经在最左边了, 那就调用 G10END。
- E10DELETE 用当前字符右边的一个字符来取代当前字符, 并且左移所有其他右边的

字符。右边最远的字符用一个空格填补。

- F10HOME 把光标设置在最左边。
- G10END 把光标设置在最右边。
- H10CHARS 绕过任一个低于 20H 或高于 7EH 的字符；如果按<Home>键，则把光标置向左边；如果按<End>键，则把光标置向右边；其他情况下，取代在串和屏幕上的字符。

改进这一程序的一些方法包括：

- 在方框的窗口内显示串。
- 使<Ins>成为一个开关键，以便字符能够取代(像现在所做的那样)或插入(把右边的字符向右挤)。

这类程序有一个复杂的问题，就是要保证你修改的文本与屏幕上的数据一致。你可能要去把数据直接传送到视频显示区的实验。

11.7 CMPS: 串比较指令

CMPS 比较一个存储单元(按 DS: SI 寻址)的内容和另一个存储单元(按 ES: DI 寻址)的内容。根据方向标志，CMPS 使 SI 和 DI 加 1 或减 1(对于字节)，加 2 或减 2(对于字)，以及加 4 或减 4(对于双字)。在进行了成功地比较或当 REP 使 CX 减到 0 时，操作结束。根据比较的结果，该操作设置 AF, CF, OF, PF, SF, 以及 ZF 标志(如果有的话)。

当 CMPS 和 REPnn 前缀相组合时，并且长度是在 CX 中，它可以比较任意长度的串。3 个 CMPS 操作对于字节是 CMPSB，对于字是 CMPSW，对于双字是 CMPSD。

注意，CMP 是操作数 2 比较操作数 1，而 CMPS 则是操作数 1 比较操作数 2。另外，CMPS 提供的是字母数字的比较，即按 ASCII 值进行比较。该操作不适于代数比较，因为它们是由带符号的数字值组成的。

考虑比较包含“Jean”和“Joan”的两个串。从左到右进行比较，每次一个字节，结果如下：

```
J: J    相等
e: o    不相等 (e 是低的)
a: a    相等
n: n    相等
```

完整的 4 个字节的比较是随着“n”和“n”的比较而结束的(相等)。现在由于两个名字是不相同的，所以在两个不同的字符之间做比较时，操作就应该立刻结束。为此目的，REP 的变种 REPE(相等时重复)只要所做的是两个相等字符之间的比较，操作就重复进行，或者直到 CX 减到 0 为止。重复单个字节比较的编码是 REPE CMPSB。

下面是使用 CMPSB 的两个例子。第一个例子比较 STRING1 和 STRING2，它们包含相同的 12 个字节值。因此，CMPSB 操作要为完整的 12 个字节而继续下去。在执行结束时，CX 的内容是 0，DI 的内容是 STRING2+12 的地址，SI 的内容是 STRING1+12 的地址，符号标志是正的，并且零标志指明相等/零。


```

STRING1 DB 'Interstellar' ; 数据项
STRING2 DB 'Interstellar'
STRING3 DB 12 DUP(' ')
...
CLD ; 从左到右
MOV CX, 12 ; 为 12 字节初始化
LEA DI, STRING2 ; ES: DI
LEA SI, STRING1 ; DS: SI
REPE CMPSB ; 比较 STRING1: STRING2
JNE exit ; 不相等, 绕过
... ; 相等

```

第二个例子比较 `STRING2` 和 `STRING3`, 它们含有不同的值。`CMPSB` 操作在比较第一个字节之后就结束, 并且结果形成高于/不等于条件; `CX` 的内容是 11, `DI` 的内容是 `STRING3+1` 的地址, `SI` 的内容是 `STRING2+1` 的地址, 符号标志是正的, 并且零标志指明不相等。

```

MOV CX, 12 ; 为 12 字节初始化
LEA DI, STRING3
LEA SI, STRING2
REPE CMPSB ; 比较 STRING2: STRING3
JE exit ; 相等, 退出
... ; 不相等

```

警告! 这些例子使用 `CMPSB` 每次一个字节地比较数据。如果你使用 `CMPSW` 每次以字为单位比较数据, 那么你就必须把 `CX` 初始化为 5, 但那是不成问题的。当对字进行比较时, `CMPSW` 把字节顺序反向。例如, 让我们比较名字 `SAMUEL` 和 `ARNOLD`。对于开始比较的字, 不是对 `SA` 和 `AR` 进行比较, 而是对 `AS` 和 `RA` 的比较。因此, 不是名字 `SAMUEL` 所代表的高于值, 而是错误地比较成低于值。`CMPSW` 只有所比较的字段是被定义为 `DW`, `DD`, 或 `DQ` 的无符号数值数据时(即按相反字节顺序存放的数据), 才能正确地工作。同样, 较短的串值应当用空白填满它右边的位置。

11.8 SCAS: 串扫描指令

`SCAS` 和 `CMPS` 不同, `SCAS` 是为一个指定值扫描一个串。为此目的, `SCAS` 把一个存储单元的内容(按 `ES: DI` 寻址)和 `AL`, `AX`, 或 `EAX` 的内容进行比较。根据方向标志, `SCAS` 也要 `DI` 加 1 或减 1(对于字节), 加 2 或减 2(对于字), 以及加 4 或减 4(对于双字)。在一次成功地比较或当 `REP` 把 `CX` 减到零时, 操作结束。`SCAS` 根据比较的结果, 置 `AF`, `CF`, `OF`, `PF`, `SF` 以及 `ZF` 标志。当 `SCAS` 与 `REPnn` 前缀相结合并且 `CX` 中是串长时, 实际上 `SCAS` 可以扫描任意长度的串。3 种 `SCAS` 操作是 `SCASB`(对于字节), `SCASW`(对于字), 以及 `SCASD`(对于双字)。

`SCAS` 特别适用于文本编辑, 在那种场合下, 程序必须要扫描标点符号, 比如句号、冒号和空格等等。

下面是为小写字母 `r` 扫描 `STRING1` 的例子。因为 `SCASB` 操作在比较不相等或直到 `CX`

为 0 之前是连续扫描的, 所以在这种情况下使用操作 **REPNE SCASB**:

```

STRING1 DB 'Intersteiliar'      ; 数据项
...
CLD                                ; 从左向右
MOV AL, 'r'                        ; 为 'r' 扫描 STRING1
MOV CX, 12                          ; 12 个字符
LEA DI, STRING1                    ; ES: DI
REPNE SCASB
JE exit                            ; 找到
...                                ; 未找到

```

当扫描在 **STRING1** 中的串“Interstellar”时, **SCASB** 在第 5 次比较时发现了匹配的情况。使用 **DEBUG** 去跟踪指令, 是在 **REP SCASB** 操作执行结束时, 这一情况才被揭示。这时, 零标志是 0, **CX** 减到 7, 而 **DI** 则加了 5(**DI** 在过了相匹配字符的实际单元后, 增加了 1 个字节)。

对于字操作, **SCASW** 为一个值扫描存储器中的一个串, 那个值与 **AX** 寄存器的字相匹配。如果使用 **LODSW** 或 **MOV** 把一个字传送到 **AX**, 那么第一个字节应该在 **AL** 中, 而第二个字节是在 **AH** 中。因为 **SCASW** 是按相反顺序比较字节的, 所以操作正确进行。

例: 使用扫描与替换

你有可能希望用另一个字符替换一个特定的字符, 例如, 要清除诸如来自一个文档中的小段和页的结束符号等编辑字符。下面的例子是为一个星号(*)扫描 **TESTDATA**, 并且用一个空格替换这个星号。如果 **SCASB** 判明了一个星号, 它便结束操作。**TESTDATA** 在 **TESTDATA+5** 处包含一个星号, 在那里插入空格, 尽管 **SCASB** 已使 **DI** 增加到 **TESTDATA+6**。**DI** 减 1, **[DI-1]** 为插入空格替换字符提供了正确的地址。

```

DATALEN EQU 13                      ; TESTDATA 的长度
TESTDATA DB 'Extra*innings'        ;
...
CLD                                ; 从左到右
MOV AL, '*'                        ; 查找字符
MOV CX, DATALEN                   ; TESTDATA 的长度
LEA DI, TESTDATA                   ; TESTDATA 的地址 (ES: DI)
REPNE SCASB                        ; 扫描 TESTDATA
JNE exit                          ; 字符找到了吗?
MOV BYTE PTR[DI-1], 20H            ; 是, 用空格替换

```

11.9 串指令的另一种编码

正如以前所讨论的, 如果串指令是明确地带后缀 **B**、**W** 或 **D** 进行编码的(比如 **MOVSB**、**MOVSW** 或 **MOVSD**), 则汇编程序会采用正确的长度并且不需要操作数。你还可以使用串操作的基本指令格式。对于像 **MOVS** 这样的指令, 它没有后缀去指明是字节、字或双字, 那么

操作数就必须指明长度。例如，如果是 **CHAR1** 和 **CHAR2** 都定义为 **DB**，则指令

```
REP MOVSB CHAR1, CHAR2
```

的意思就是重复地传送从 **CHAR2** 开始的字节到开始于 **CHAR1** 的字节。

另外一种格式允许你明确地指定段寄存器和使用 **PTR** 伪操作。如果你把 **CHAR1** 和 **CHAR2** 的地址装入 **DI** 和 **SI** 寄存器，那么就可以把 **MOVSB** 指令编写成

```
LEA DI, CHAR2
LEA SI, CHAR1
REP MOVSB: BYTE PTR[DI], DS:[SI]
```

很少有程序会被编成这样，并且在这里说明这些格式也仅仅是作为一种资料而已。

11.10 复制一种模式

STOS 指令对于用指定的字节、字、或双字的值去设置一个区域是有用的。但是，对于超过这些长度而去重复一种模式来说，可以使用作了局部修改的 **MOVSB**。假定希望建立一个以下模式的显示行：

```
|***||***||***||***||***| ...
```

你不需要重复定义整个模式，而只需要紧靠在显示行之前定义前 6 个字节。下面是一些指令：

PATTERN	DB	' *** '		: 模式，紧靠在前面
DISPAREA	DB	42 DUP(?)		: 显示区
...				
	CWD			: 从左到右操作
	MOV	CX, 21		: 21 个字
	LEA	DI, DISPAREA		: 目的
	LEA	SI, PATTERN		: 源
	REP	MOVSB		: 传送字符

在执行时，**MOVSB** 把 **PATTERN** 的第一个字(*)传送到 **DISPAREA** 的第一个字，然后传送第二个字(**)和第三个字(*):

```

|***||***|
  ↑   ↑
PATTERN DISPAREA
```

在这时，**DI** 包含 **DISPAREA+6** 的地址，而 **SI** 包含 **PATTERN+6** 的地址，这也是和 **DISPAREA** 一样的地址。现在，操作自动地复制这种模式，把 **DISPAREA** 的第一个字传送到 **DISPAREA+6**，把 **DISPAREA+2** 传送到 **DISPAREA+8**，把 **DISPAREA+4** 传送到 **DISPAREA+10**，以此类推。最后，该模式一直复制到 **DISPAREA** 的末端：

```

|***||***||***||***||***| ... |***|
  ↑       ↑       ↑       ↑
PATTERN  DISPAREA+6  DISPAREA+12  DISPAREA+42
```

你可以使用这一技术任意次地复制一种模式。该模式本身可以是任意长，但必须紧靠在目的字段的前面。

11.11 要 点

- 对于串指令 **MOVS**, **STOS**, **CMPS**, 以及 **SCAS**, **EXE** 程序必须初始化 **ES** 寄存器。
- 串指令使用后缀 **B**、**W** 或 **D** 去处理字节串、字串或双字串。
- 方向标志控制处理所要求的方向：清除标志为从左到右的方向，设置标志(**STD**)为从右到左的方向。
- 对于 **MOVS**, **DI** 和 **SI** 控制操作数寻址，而对于 **CMPS**, **SI** 和 **DI** 则是控制寻址。
- **CX** 必须包含初始值，以便 **REP** 去处理指定数量的字节、字或双字。
- 对于正常的处理，**REP** 是和 **MOVS** 与 **STOS** 一起使用的，而条件 **REP**(**REPE** 或 **REPNE**) 是和 **CMPS** 与 **SCAS** 一起使用的。
- **CMPSW** 和 **SCASW** 操作在被比较的字中，字节顺序是反的。
- 为了从右到左处理，是从字段的最右边的字节开始寻址的。例如，如果命名为 **COTITLE** 的字段是 10 个字节长，那么对于处理字节，**LEA** 装入地址是 **COTITLE+9**。对于处理字，装入地址是 **COTITLE+8**，这是因为串操作最初是访问 **COTITLE+8** 和 **COTITLE+9**。

11.12 习 题

11-1. 串操作采用了与 **ES:DI** 或 **DS:SI** 寄存器有关的操作数。请指出用于以下串操作的寄存器：(a)**MOVS**(操作数 1 和操作数 2)，(b)**LODS**(操作数 1)，(c)**CMPS**(操作数 1 和操作数 2)，(d)**STOS**(操作数 2)。

11-2. 对于属于 **REP MOVSB** 一类的串操作(a)如何设置重复次数?(b)如何设置从右到左的处理?

11-3. 本章给出了与以下一些带 **REP** 前缀的指令等效的指令：(a)**MOVSB**, (b)**LODSB**, (c)**STOSB**。对于以上每种情况，请提供处理字的等效代码。

11-4. 修改图 11-2 的程序，改变 **MOVSB** 和 **MOVSW** 操作从右到左传送数据。使用调试程序跟踪整个过程，并注意数据段和寄存器的内容。

11-5. 以下各条指令要测试什么条件：(a)**REPNE CMPSW**, (b)**REPE SCASW**。

11-6. 使用下面的数据定义并按(a)到(f)的要求编写不相关的串操作指令：

```
BUS_TITLE    DB    'Computer Wizards'
WORK_SPACE   DB    16 DUP(20H)
```

(a) 从左到右，把 **BUS_TITLE** 传送到 **WORK_SPACE**。

(b) 从右到左，把 **BUS_TITLE** 传送到 **WORK_SPACE**。

- (c) 把 `BUS_TITLE` 的第四个和第五个字节装入 `AX`。
- (d) 把开始于 `WORK_SPACE+12` 的字符存入 `AX`。
- (e) 比较 `BUS_TITLE` 和 `WORK_SPACE`。
- (f) 为第一个空格字符扫描 `BUS_TITLE`，如果找到的话，把它传送到 `CH`。

11-7. 修改在“SCAS: 串扫描指令”一节的程序段，使该操作为字符对 `st` 而去扫描 `STRING1`。`STRING1` 的检查发现 `st` 没有作为一个字出现，就像下面：`/In/te/rs/te/ll/ar/`那样。两种可能解决的方案是：(a)使用 `SCASW` 两次，第一个 `SCASW` 从 `STRING1` 开始，而第二个 `SCASW` 则是从 `STRING1+1` 开始；(b)或使用 `SCASB` 并在找到 `s` 时，比较紧跟它后面的字节是不是 `t`。

11-8. 定义含有十六进制值 `C9CDCDCDBB` 的一个 5 字节字段。用 `MOVSB` 把这个字段复制 12 次成为一个 60 个字节的区域,并显示其结果。

11-9. 按以下要求编程序。用串“Computer Tech”定义 `NAME1`，并用 13 个空格定义 `NAME2`。使用 `LODSB` 从左到右去取 `NAME1` 中的每个字符。然后使用 `STOSB` 把每个所取到的字符从右到左存入 `NAME2`，使 `NAME2` 所包含的串是按相反顺序的。为这一过程清除和设置方向标志，汇编这个程序并测试它。

11-10. 修改图 11-3 去检查 `Ins` 键。如果它是关断的，则所输入的字符把现有的字符覆盖掉；如果它是接通的，则插入该字符，使其右边的字符都从左向右挤，最右边的字符被删除。汇编并测试这一程序。

算术运算 I :

处理二进制数据

目的：介绍二进制数据的加法，减法，乘法与除法的要求

12.1 引言

这一章涉及加法，减法，乘法，除法，以及无符号与带符号数值数据的使用，还包括许多例子和对于在计算机算术运算领域里粗心大意的朋友所犯各种错误的警告。第 13 章会涉及到一些特殊的要求，包括二进制与 ASCII 数据格式之间的转换。

虽然我们习惯于做十进制(基数为 10)格式的算术运算，但微计算机却只能做二进制(基数为 2)算术运算。本章所要说明的指令是：

ADC	带进位加法	IDIV	带符号数除法
ADD	加法	IMUL	带符号数乘法
CBW	字节转换为字	MUL	无符号数乘法
CDQ	双字转换为四字	NEG	求反
CWD	字转换为双字	SBB	带借位减法
CWDE	字转换为扩展双字	SUB	减法
DIV	无符号数除法		

12.2 处理无符号与带符号的二进制数据

某些数值字段，例如用户编号和日期是无符号的。某些带符号的数值字段，例如用户该付的差额，温度的读数可能包含正值或负值。其他的带符号数值字段，例如产品价格和 π 值永远都被认定为正值。

下面的表给出与寄存器宽度相匹配的无符号和带符号数据的最大值：

格式	字节	字	双字
无符号	255	65535	4294967295
带符号	127	32767	2147483647

对于无符号数据,所有位都用作数据位;对于带符号数据,最左边的位是符号位。但是注意,ADD 和 SUB 指令是不区分无符号和带符号数据的,的确是简单地对各位做加法和减法。

下面的例子说明 2 个二进制数的加法,这些值可以表示为无符号数和带符号数。上面的数左边有为 1 的位,对于无符号数据,这些位表示 249,而对于带符号数据,这些位表示-7。该加法使溢出(OF)或进位(CF)标志均置 0:

二进制	无符号数 十进制	带符号数 十进制	OF	CF
11111001	249	-7		
+ 00000010	+ 2	+ 2		
11111011	251	-5	0	0

对于无符号与带符号数据来说,二进制加法的结果是一样的。但是,在无符号字段中的这些位表示的是十进制的 251,而在带符号的字段中,它们表示十进制-5。实际上,一个字段的內容是任何你打算让它们表示的内容,并且必须对它们进行相应的处理。

1. 算术进位。算术操作把符号位所产生的(结果 0 或 1)传送到进位标志。如果该符号位是 1,则实际是把进位标志置成 1。若无符号数据中产生了进位,其运算结果是无效的。以下是产生进位的加法举例:

二进制	无符号数据 十进制	带符号数据 十进制	OF	CF
11111101	252	-4		
+ 00000101	+ 5	+ 5		
(1)00000001	1	1	0	1
	(无效的)	(有效的)		

对于无符号数据的操作是无效的,原因在于数据位的进位输出,而对于带符号数据的操作则是有效的。

2. 算术溢出。当符号位有进位输入而又没有进位输出,或者进位输出不是由进位输入产生时,算术操作将把溢出标志置成 1。如果溢出是发生在带符号数据中,则运算结果就是无效的(因为溢出进入了符号位)。以下是产生溢出的加法举例:

二进制	无符号数据 十进制	带符号数据 十进制	OF	CF
01111001	121	+121		
+ 00001011	+ 11	+ 11		
10000100	132	-124	1	0
	(有效的)	(无效的)		

加法操作可以同时把进位标志与溢出标志都置成 1。在下面的例子中,进位使得无符号数据的操作是无效的,而溢出则使带符号数据的操作是无效的:

二进制	无符号数据 十进制	带符号数据 十进制	OF	CF
11110110	246	-10		
+ 10001001	+ 137	- 119		
(1)01111111	127	+ 127	1	1
	(无效的)	(无效的)		

所有这一切的结论是：必须弄清楚程序将要处理的数的大小，并要相应地定义与处理这些数字字段。

12.3 二进制数据的加法与减法

ADD 和 SUB 指令按照字节、字和双字的大小执行简单的二进制数据的加法和减法。格式是：

[Label:]	ADD/SUB	register, register
[Label:]	ADD/SUB	memory, register
[Label:]	ADD/SUB	register, memory
[Label:]	ADD/SUB	register, immediate
[Label:]	ADD/SUB	memory, immediate

ADD 或 SUB 操作设置或清除溢出与进位标志，正如上一节所讨论的，跟其他大多数指令一样，没有直接的存储器到存储器的操作。

如第 1 章所述，负的二进制数是用二进制补码来表示的，它是把正数按位求反再加 1 而形成的。下面自解释性的 ADD 与 SUB 举例是处理字节值和字值的。

```

BYTE1    DB    24H           ; 数据项
WORD1     DW    4000H
...
MOV  CL, BYTE1                ; 处理字节:
MOV  DL, 40H
ADD  CL, DL                    ; 寄存器到寄存器
SUB  CL, 20H                    ; 从寄存器减去立即数
ADD  BYTE1, BL                  ; 寄存器到存储器
MOV  CX, WORD1                 ; 处理字:
MOV  DX, 2000H
SUB  CX, DX                     ; 从寄存器减去寄存器
SUB  CX, 124H                   ; 从寄存器减去立即数
ADD  WORD1, DX                  ; 寄存器加到存储器

```

12.3.1 溢出

溢出在算术操作，尤其在带符号数据算术操作方面，是值得关注的一件事情。由于二进制

12.3.2 字节扩展为字

上一节说明了 20H 和在 AL 中的 60H 相加如何产生了不正确的和, 一个较好的解决办法是在 AX 中执行算术运算。为此目的, 要用到的指令是 CBW(把字节转换为字), 它会自动地扩展 AL 的符号位(0 或 1)到整个 AH。注意, CBW 没有操作数, 它只限于使用 AX。

在下一个例子中, CBW 把在 AL 中的符号位(0)扩展到整个 AH, 在 AX 中产生 0060H, 然后该例子把 20H 加到 AX(不是加到 AL)并且在 AX 中产生正确的结果: 0080H 或 +128:

...		AH	AL
		xx	60H
CBW	: 把 AL 的符号扩展到 AH	00	60
ADD AX, 20H	: 加到 AX	00	80

这个例子的数值结果和上一个例子的结果是一样的, 但是在 AX 中的加法不会把它当作溢出或负数来处理。另外, 即使 AX 中的整个字允许 1 个符号位和 15 个数据位, 但 AX 对值的限制还是 -32768 到 +32767。

其他把字节转换成字的指令是 MOVZX(对于无符号数据)和 MOVSX(对于带符号数据), 像以下这样使用:

MOVZX CX, BYTEVAL	: 字节在 CL 中, 零在 CH 中
MOVSX WORDVAL, DL	: 字节送到字中, 用符号填充高位字节

12.3.3 字扩展为双字

CWD(字转换为双字)指令用于把一个带符号的数的字扩展为双字, 方法是把 AX 的符号位复制到整个 DX。注意, CWD 是没有操作数的, 它只限于使用 DX:AX。这里是一个例子:

MOV AX, WORD	: 把字传送到 AX
CWD	: 把字扩展到 DX:AX

CWDE(字转换为扩展的双字)指令用于把一个带符号数的字扩展为双字, 方法是把 AX 的符号位复制到整个 EAX。这里是一个例子:

MOV AX, WORD1	: 把字传送到 AX
CWDE	: 把字扩展到 EAX

MOVZX 和 MOVSX 也可以把字转换成双字:

MOVZX ECX, WORDVAL	: 字在 CX 中, 用零填充 ECX 的高位字
MOVSX DBLWORD, DX	: 字在 DBLWORD 中, 用符号填充其高位字

CDQ(双字转换为四字)指令用于把带符号数的双字扩展成四字, 方法是把 EAX 的符号位复制到 EDX。注意, CDQ 是没有操作数的, 因而它只限于使用 EDX:EAX。这里是一个例子:

```
MOV EAX, DBLWORD      ; 把双字传送到 EAX
CDQ                    ; 把双字扩展到 EDX:EAX
```

12.3.4 实现双字值的算术运算

正如我们已经看到的，大的数值可能超过一个字的容量，而需要多字的容量。在多字运算方面，一个需要考虑的问题是：相反的字节顺序和相反的字顺序。回想一下，汇编程序自动地把所定义的数值字的内容转换成按相反字节顺序排列，例如 0134H 的定义是按 3401H 存放的。下面的例子是双字值的相加与存放：

```
DBLWORD1 DD 0123BC62H      ; 定义双字
DBLWORD2 DD 0012553AH
DBLWORD3 DD 0
...
MOV ECX, DBLWORD1          ; 加与存放
ADD EAX, DBLWORD2          ; 双字
MOV DBLWORD3, EAX          ; 值
```

汇编程序自动地按相反字节(和字)顺序排列所定义的数据。但对于某些应用场合，数据可能被定义成字值。对于在上一例子中所定义的 DBLWORD1 来说，必须把字按相邻相反的次序来定义：

```
DW 0BC62H
DW 0123H
```

然后汇编程序把这些定义按相反字节顺序转换为 162BC12301H，现在就适合于作双字算术运算了。下面考察一下对这些值执行算术运算的两种方法。第一种是简单的和专用的，而第二种则是比较复杂的和通用的。下面是该例子所使用的数据：

```
WORD1A DW 0BC62H      ; 数据项
WORD1B DW 0123H
WORD2A DW 553AH
WORD2B DW C012H
WORD3A DW ?
WORD3B DW ?
```

实际上，程序是要把如下值相加：

```
WORD1B: WORD1A  0123 BC62H
WORD2B: WORD2A  0012 553AH
-----
WORD3B: WORD3A  0136 119CH
```

由于在存储器中是按相反字节顺序的，程序用相邻的但又是相反的字，分别来定义值：BC62 0123 和 553A 0012。然后，汇编程序把这些双字值按相反字节顺序存放在存储器中：

```
WORD1A 和 WORD1B:  62BC 2301
WORD2A 和 WORD2B:  3A55 1200
```

第一个例子是把第一对字(WORD1A 和 WORD1B)和第二对字(WORD2A 和 WORD2B)

相加，并把和存放在第三对字(WORD3A 和 WORD3B)中：

```

MOV  AX, WORD1A      ; 加最左边的字
ADD  AX, WORD2A
MOV  WORD3A, AX
MOV  AX, WORD1B      ; 加最右边的字
ADC  AX, WORD2B      ; 带进位
MOV  WORD3B, AX

```

该例子首先把 WORD2A 加到 AX 中的 WORD1A(低阶部分)上，并把和存放在 WORD3A 中。下一步是把 WORD2B 及上一次加法的进位一起加到 AX 中的 WORD1B(高阶部分)上，然后把和存放在 WORD3B 中。

让我们仔细观察一下这些操作：第一组 MOV 和 ADD 操作在 AX 中的字节是相反顺序的，并且使最左边的字相加(先加低阶字)：

```

WORD1A:      BC62H
WORD2A:      + 553AH
-----
总计          (1)119CH  (9C11 被存放在 WORD3A 中)

```

由于 WORD1A 加 WORD2A 的和超过了 AX 的容量，产生进位，置进位标志为 1。接下来右边的字相加，但这次是用 ADC(带进位加)代替 ADD。ADC 把 2 个值相加，因为进位标志置 1，所以和要加 1(在加高阶字时要用 ADC，以便计入低阶字相加所产生的进位。):

```

WORD1B      0123H
WORD2B      + 0012H
加上进位    + 1H
-----
总计          0136H  (按 3601H 存放在 WORD3B 中)

```

利用调试程序去跟踪该算术运算展示在 AX 中的和，在 WORD3A 中是相反字节顺序值 9C11H，以及在 WORD3B 中是 3601H。

第二个例子提供了一个比较复杂的方法把任意长度值相加，这里相加的还是以前一样的字对：WORD1A:WORD1B 和 WORD2A:WORD2B：

```

CLC                      ; 清除进位标志
MOV  CX, 02              ; 设置循环计数
LEA  SI, WORD1A          ; 字对的最左边的字
LEA  DI, WORD2A          ; 字对的最左边的字
LEA  BX, WORD3A          ; 和的最左边的字
L20:
MOV  AX, [SI]            ; 把字传送到 AX
ADC  AX, [DI]            ; 带进位加到 AX
MOV  [BX], AX            ; 存储字
INC  SI                  ; 为下一个
INC  SI                  ; 向右的字
INC  DI                  ; 调整地址
INC  DI
INC  BX
INC  BX

```

LOOP L20

；为下一个字重复

...

该例子用 SI、DI 和 BX 分别作为 WORD1A、WORD2A 和 WORD3A 的地址的变址寄存器。对于每个要相加的字对都要经过所有指令循环一次在这种情况下，是两次。第一次循环把最左边的字相加，而第二次循环则是把最右边的字相加。由于第二个循环是向右处理字的，所以在 SI、DI 和 BX 中的地址要加 2。两条 INC 指令对于每个寄存器执行这一操作。使用 INC(而不是 ADD)的一个良好理由是：指令 ADD reg, 02 将清除进位标志，并且会产生一个不正确的答案；相反，INC 不影响进位标志。

由于循环，所以只有一条加法指令 ADC。在开始时，CLC(清除进位)指令确保进位标志最初是被清除的。在 WORD3A、WORD3B 和 AX 中的结果和前面的例子是一样的。

为了按这种方法工作，必须(1)定义的字是彼此相邻的，(2)把 CX 初始化为要相加的字数，以及(3)从左到右处理字。

对于多字减法，与 ADC 相当的指令是 SBB(带借位减法)。用 SBB 简单地取代 ADC。这里是 ADC 和 SBB 的格式：

{label:}	ADC/SBB	register, register
{label:}	ADC/SBB	memory, register
{label:}	ADC/SBB	register, memory
{label:}	ADC/SBB	register, immediate
{label:}	ADC/SBB	memory, immediate

对于四字相加，使用前面所述对于多字相加的技术，定义 2 个相邻的双字对，并使用 ECX 寄存器。

12.4 二进制数据乘法

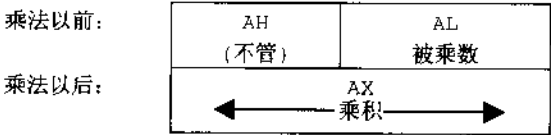
对于乘法，MUL(乘)指令用于处理无符号数据，而 IMUL(整数乘)则用于处理带符号数据。两条指令都影响进位标志和溢出标志。作为程序员，应该全面控制所处理的数据格式，而且有选择合适的乘法指令的责任。MUL 和基本的 IMUL 的格式是：

{label: }	MUL/IMUL	register, memory
-----------	----------	------------------

乘法操作是字节乘字节，字乘字或双字乘双字。IMUL 提供 3 种附加格式，它们包含双字和立即操作数，这在下一节将会涉及到。

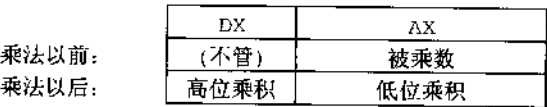
12.4.1 字节乘字节

对于 2 个一字节值相乘，被乘数在 AL 寄存器中，而乘数是在存储器或另一个寄存器中的一个字节。对于指令 MUL DL，该操作是用 DL 的内容去乘 AL 的内容，所产生的乘积在 AX 中。该操作可能覆盖已经在 AH 中的任何数据。



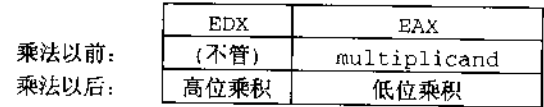
12.4.2 字乘字

对于 2 个一字值相乘，被乘数放在 AX，乘数是在存储器或另一个寄存器中的一个字。对于指令 MUL DX，该操作是用 DX 的内容去乘 AX 的内容。所产生的乘积是一个双字，需要两个寄存器：高位(最左边)部分在 DX 中，低位(最右边)部分在 AX 中。该操作可能覆盖已在 DX 中的任何数据。



12.4.3 双字乘双字

对于 2 个双字值相乘，被乘数在 EAX，乘数是在存储器或另一个寄存器中,所产生的乘积在 EDX:EAX 对中。该操作可能覆盖已在 EDX 中的任何数据。



12.4.4 字段大小

MUL 或 IMUL 的操作数只引用乘数，由它决定字段大小。指令设定被乘数是在 AL, AX, 或 EAX 中，这取决于乘数的大小。在以下的例子中，乘数是在一个寄存器中的字节、字或双字：

指令	乘数	被乘数	乘积
MUL CL	字节	AL	AX
MUL BX	字	AX	DX:AX
MUL EBX	双字	EAX	EDX:EAX

在下面的例子中，乘数是在存储器中定义的：

BYTE	DB	?	; 字节值
WORD1	DW	?	; 字值
DWORD1	DD	?	; 双字值

指令	乘数	被乘数	乘积
MUL BYTE1	BYTE1	AL	AX
MUL WORD1	WORD1	AX	DX:AX
MUL DWORD1	DWORD1	EAX	EDX:EAX

12.4.5 无符号数乘法：MUL

MUL 指令的用途是进行无符号数据相乘。以下程序片段给出 4 个使用 MUL 的例子：字节×字节，字×字，字×字节，以及双字×双字。数据定义如下：

```

BYTE1      DB      80H
BYTE2      DB      40H
WORD1      DW      8000H
WORD2      DW      2000H
DWORD1     DD      00018402H
DWORD2     DD      00012501H

```

例 1 40H 乘 80H。在 AX 中的乘积是 2000H(8192)：

```

MOV AL, BYTE1      ; 字节×字节
MUL BYTE2          ; 乘积在 AX 中

```

例 2 在 DX: AX 对中产生 1000 0000H：

```

MOV AX, WORD1      ; 字×字
MUL WORD2          ; 乘积在 DX: AX 中

```

例 3 字和字节相乘并要求把 BYTE1 扩展成一个字：

```

MOVZX AX, BYTE1    ; 字节×字
MUL WORD1          ; 乘积在 DX: AX 中

```

由于值是被假定为无符号的，所以该例子认为在 AH 中的位是 0。(这里使用 CBW 可能引发的问题是，AL 最左边的位可能是 1，并且在 AH 中扩展为 1 的位会导致一个较大的无符号的值)。在 DX: AX 中的积是 0040 0000H。

例 4 使用 EAX 进行双字相乘：

```

MOV EAX, DWORD1    ; 双字×
MUL DWORD2          ; 双字

```

在 EDX: EAX 中的值是 00000001 BC17CE02H。

12.4.6 带符号数乘法：IMUL

IMUL 指令的用途是进行带符号数据的相乘。以下的程序片段表示的和刚给出的 4 个例子是一样的，只是用 IMUL 替代了 MUL。

例 1 80H(一个负数)乘以 40H(一个正数)：

```

MOV AL, BYTE1      ; 字节×字节
IMUL BYTE2         ; 乘积在 AX 中

```

在 AX 中的乘积是 E000H。使用同样的数据时，MUL 产生的乘积是 2000H，所以可以看出使用 MUL 和使用 IMUL 的区别。MUL 把 80H 当成+128，而 IMUL 则把 80H 当成-128。-128 乘+64 的乘积是-8192H，等于 E000H(尝试一下把 E000H 转换成位，这些位求反，加 1，再加

上位的权值)。

例 2 8000H(一个负值)乘以 2000H(一个正值)。在 DX: AX 中的乘积是 F000 0000H, 这是 MUL 所产生的乘积的负值:

```
MOV AX, WORD1      ; 字×字
IMUL WORD2          ; 乘积在 DX: AX 中
```

例 3 首先把 BYTE1 扩展为 AX 中的一个字。因为已假定该值是带符号数, 所以例子使用 MOVSX 把最左边的符号位扩展到 AH: AL(AL=80H), 变成在 AX 中的 FF80H。由于乘数 WORD1 也是负的, 所以乘积应该是正的。并且在 DX: AX 中它确实是正的 0040 0000H——和 MUL 2 个无符号数相乘的结果是一样的。

```
MOVSX AX, BYTE1     ; 字节×字
IMUL WORD1           ; 乘积在 DX: AX 中
```

例 4 使用 EAX 进行双字相乘。乘积在 EDX: EAX 中, 其值为 00000001 BC17CE02H:

```
MOV EAX, DWORD1     ; 双字×
IMUL DWORD2          ; 双字
```

实际上, 如果被乘数与乘数有相同的符号位, 那么 MUL 和 IMUL 就产生相同的乘积。但如果被乘数与乘数符号位不相同, 则 MUL 产生一个正的乘积, 而 IMUL 产生一个负的乘积。结论是: 程序必须要知道数据的格式并使用相应的指令。
试一下用调试程序去跟踪全部例子。

12.4.7 其他 IMUL 格式

80286 和 80386 处理器引入了 3 种 IMUL 格式, 提供立即操作数以及产生乘积在 AX 以外的寄存器中。可以把它们用于无符号数和带符号数的乘法, 因为结果是一样的。这些值必须都是同样长度的: 16 位或 32 位。以下是它们的格式:

16 位立即数	[label:]	IMUL register, immediate
32 位立即数	[label:]	IMUL register, memory, immediate
16/32 位	[label:]	IMUL register, register/memory

- 1. 16 位立即数 IMUL 操作。第一个操作数(寄存器)包含被乘数, 而第二个操作数(立即值)是乘数, 所产生的乘积在第一个操作数中。超出寄存器的乘积使进位标志和溢出标志被置 1。
- 2. 32 位立即数 IMUL 操作。在双字格式下, 这一格式有 3 个操作数。第二个操作数(存储器)包含被乘数, 第三个操作数(立即值)包含乘数, 所产生的乘积在第一个操作数(寄存器)中。
- 3. 16/32 位 IMUL 操作。第一个操作数(寄存器)包含被乘数, 第二个操作数(寄存器/存储器)包含乘数, 所产生的乘积放在第一个操作数中。

下面是这 3 种 IMUL 指令的例子:

规模	指令	被乘数	乘数	乘积
16 位立即数	IMUL DX, 25	DX	25	DX

32 位立即数	IMUL ECX, MULTCAND, 25	MULTCAND	25	ECX
16/32 位	IMUL BX, CX	BX	CX	BX
16/32 位	IMUL EBX, EDX	EBX	EDX	EBX
16/32 位	IMUL EBX, DWORDVAL	EBX	DWORDVAL	EBX

12.4.8 执行双字乘法

一般的乘法包括字节乘字节，字乘字，双字乘双字。正如我们已经见到的，字的最大带符号值是+32767。80386 之前的处理器处理更大值的相乘要包括一些附加的步骤。这些处理器所用的方法是每个字分别相乘，然后再把每个乘积相加。这样的例子是值得研究的，因为这一技术可以用于双字寄存器，如 EAX。下面的例子是 4 位的十进制数乘以 2 个数位的数：

$$\begin{array}{r} 1365 \\ \times 12 \\ \hline 16380 \end{array}$$

如果只进行 2 个数位的数相乘是怎样的呢？可以将 13 和 65 分别乘以 12，就像这样：

$$\begin{array}{r} 13 \\ \times 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 65 \\ \times 12 \\ \hline 780 \end{array}$$

接下来，把两个乘积相加；但请记住，由于 13 是在百的位置，它的乘积实际上是 15600：

$$\begin{array}{r} 15600 \quad (13 \times 12 \times 100) \\ + 780 \quad (65 \times 12) \\ \hline 16380 \end{array}$$

汇编语言程序可以使用同样的技术，除非数据是由十六进制格式的字(4 个数位)组成的。下一节探讨双字与字相乘及双字与双字相乘的一些要求。

1. 双字乘以字。下面的程序片段是一个双字乘以字。被乘数 MULTCAN 分别由 2 个字 3206H 和 2521H 组成。定义 2 个 DW 代替定义 DD 的原因是为了便于 MOV 指令的寻址，该 MOV 指令是把字传送到 AX 的。该值按相反字节顺序定义，并且汇编程序是按相反字节顺序存放每个字的。因此，MULTCAN 被定义的值是 32062521H，按 21250632H 存放。

```
MULTCAN    DW    2521H      ; 数据项
            DW    3206H
MULTPLR     DW    6400H
PRODUCT     DW    0, 0, 0    ; 3 个字
```

乘数 MULTPLR 的内容是 6400H。为所产生的乘积提供了 3 个字的字段 PRODUCT。第一个 MUL 操作是把 MULTPLR 和 MULTCAN 左边的字相乘，乘积是十六进制的 0E80 E400H，存放在 PRODUCT+2 和 PRODUCT 中：

```
MOV AX, MULTCAN      ; 乘被乘数
MUL MULTPLR          ; 左边的字(低阶字)
MOV PRODUCT, AX      ; 保存乘积
MOV PRODUCT+2, DX
```


第二个 MUL 是把 MULTPLR 和 MULTCAN 右边的字相乘, 乘积是 138A 5800H。

```
MOV AX, MULTCAN+2      ; 乘被乘数
MUL MULTPLR             ; 右边的字(高阶字)
ADD PRODUCT+2, AX       ; 加到第二个
ADC PRODUCT+4, DX       ; 和第三个字上
```

最后, 该例子把两个乘积相加, 如以下这样:

```
乘积 1:   0000 0E80 E400
乘积 2:   + 138A 5800
-----
总计      138A 6680 E400
```

由于第一个 ADD 可能产生进位, 所以第二个加法是 ADC(带进位加)。数值数据是按相反字节格式存放的, 所以 PRODUCT 的内容实际上是 00E4 8066 8A13。该例程序要求 PRODUCT 的第一个字最初包含零。

2. 双字乘以双字。在 DX:AX 中的 2 个双字相乘包括做 4 次乘法:

被乘数		乘数
字 2	×	字 2
字 2	×	字 1
字 1	×	字 2
字 1	×	字 1

把在 DX 和 AX 中的每个乘积加到最后乘积的适当的字上。以下的程序片断给出一个例子。MULTCAND 包含 3206 2521H, MULTPLER 包含 6400 0A26H, 而 PRODUCT 提供了 4 个字:

```
MULTCAND    DW  2521H      ; 数据项
              DW  3206H
MULTPLER     DW  0A26H
              DW  6400H
PRODUCT      DW  0, 0, 0, 0 ; 4 个字
```

虽然逻辑上是类似于字和双字相乘, 但这个问题还需要一个附加的特性。在 ADD/ADC 对之后是另一个 ADC, 它把 0 加到 PRODUCT 上。第一个 ADC 本身可能产生进位, 后续的指令会把它清除, 因此, 第二个 ADC 如果没有进位就加 0, 有进位就加 1。最后的 ADD/ADC 对不需要附加的 ADC, 因为 PRODUCT 对于最终所得到的答案是足够大的, 所以没有进位。

```
MOV AX, MULTCAND      ; 被乘数字 1(低阶字)
MUL MULTPLER           ; ×乘数字 1(低阶字)
MOV PRODUCT+0, AX      ; 存放乘积
MOV PRODUCT+2, DX
;
MOV AX, MULTCAND       ; 被乘数字 1(低阶字)
MUL MULTPLER+2         ; ×乘数字 2(高阶字)
ADD PRODUCT+2, AX      ; 加到所存的乘积上
ADC PRODUCT+4, DX
ADC PRODUCT+6, 00      ; 加任何进位
```

```

;
MOV AX, MULTCAND+2      ; 被乘数字 2 (高阶字)
MUL MULTPLER             ; × 乘数字 1 (低阶字)
ADD PRODUCT+2, AX        ; 加到所存的乘积上
ADC PRODUCT+4, DX
ADC PRODUCT+6, 00        ; 加任何进位
;
MOV AX, MULTCAND+2      ; 被乘数字 2 (高阶字)
MUL MULTPLER+2           ; × 乘数字 2 (高阶字)
ADD PRODUCT+4, AX        ; 加到乘积上
ADC PRODUCT+6, DX

```

最后的乘积是 138A 687C 8E5C CCE6，以相反的字节顺序存放在 PRODUCT 中。试一试使用调试程序去跟踪整个例子。

12.4.9 用移位做乘法

对于乘以 2 的幂(2, 4, 8 等等)的情况，左移必要的位数可以提高处理速度。在以下不相关的例子中，被乘数是在第一个操作数中，而 CL 的内容是 2：

```

乘以 2 (左移 1 次):  SHL AX, 01
乘以 4 (左移 2 次):  SHL DX, CL
乘以 8 (左移 3 次):  SHL WORDVAL, 03

```

下面的例程序可用于在 DX:AX 对中的双字值的左移。虽然指定移 4 位，但它可以适应于其他值：

```

SHL DX, 04      ; DX 左移 4 位
MOV BL, AH      ; 把 AH 存入 BL
SHL AX, 04      ; AX 左移 4 位
SHR BL, 04      ; BL 右移 4 位
OR DL, BL       ; 从 BL 往 DL 中插入 4 位

```

在移出有效数位之前，一定要进行检查。

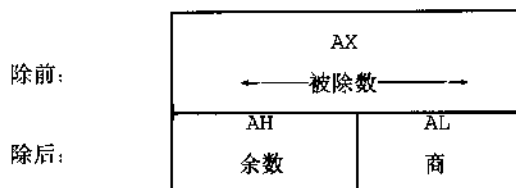
12.5 二进制数据除法

对于除法，DIV(除法)指令处理无符号数据，IDIV(整数除法)处理带符号数据。要选择适当的除法指令进行操作。DIV/IDIV 的格式是：

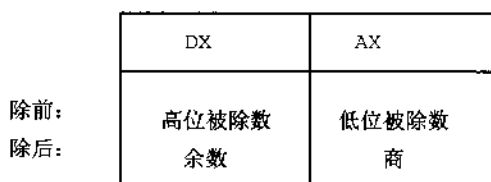
{label: }	DIV/IDIV	register/memory
-----------	----------	-----------------

基本的除法操作是字除以字节，双字除以字，以及四字除以双字。

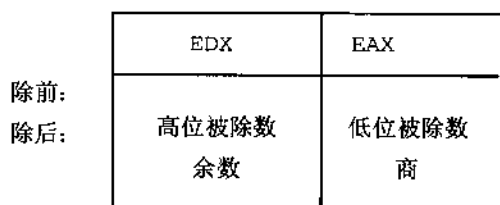
1. 字除以字节。为了用字节去除字，把被除数在 AX 中，除数是在存储器或另一寄存器中的一个字节。该操作把余数存放在 AH 中，而把商存放在 AL 中。注意，1 字节的商是很小的——如果是无符号数，则最大值是+255(FFH)；如果是带符号数，则最大值是+127(7FH)。



2. 双字除以字。为了用字去除双字，被除数在 DX:AX 中，除数是在存储器或另一寄存器中的一个字。该操作把余数存放在 DX 中，把商存放在 AX 中。一个字的商允许的最大值对于无符号数是+32767(FFFFH)，对于带符号数则是+16383(7FFFH)。



3. 四字除以双字。对于双字去除四字，被除数是在 EDX:EAX 中，除数是在存储器或另一寄存器中的一个双字。该操作把余数存放在 EDX 中，而商放在 EAX 中。



4. 字段大小。DIV/IDIV 的操作数引用除数，它决定字段的大小。在下面的 DIV 例子中，除数是在寄存器中的字节、字或双字。

指令	除数	被除数	商	余数
DIV CL	字节	AX	AL	AH
DIV CX	字	DX:AX	AX	DX
DIV EBX	双字	EDX:EAX	EAX	EDX

在下面的 DIV 例子中，除数是定义在存储器中的：

```

BYTE1    DB    ?    ; 字节值
WORD1     DW    ?    ; 字值
DWORD1    DD    ?    ; 双字值

```

指令	除数	被除数	商	余数
DIV BYTE1	BYTE1	AX	AL	AH
DIV WORD1	WORD1	DX:AX	AX	DX
DIV DWORD1	DWORD1	EDX:EAX	EAX	EDX

5. 余数。13 除以 3 的结果是 4 1/3，其中商是 4，余数是 1。注意，计算器(以及高级程序设计语言)将提供一个 4.333 的商，它是由一个整数部分(4)和小数部分(.333)组成的。值 1/3

和.333 是小数，而 1 是个余数。

12.5.1 使用 DIV 做无符号数除法

DIV 指令用于做无符号数的除法。以下程序片断给出 4 个 DIV 的例子：字除以字节，字节除以字节，双字除以字，以及字除以字。下面是例子所用的数据：

```

BYTE1    DB    80H    ; 数据项
BYTE2    DB    16H
WORD1    DW    2000H
WORD2    DW    0010H
WORD3    DW    1000H

```

例 1 是 2000H(8192)除以 80H(128)。在 AH 中的余数是 00H，而在 AL 中的商是 40H(64)：

```

MOV  AX, WORD1    ; 字/字节
DIV  BYTE1        ; 余数:商在 AH:AL 中

```

例 2 要求把 BYTE1 扩展成一个字，因为该值被假定是无符号数，所以例子假设 AH 的各位都是 0。在 AH 中的余数是 12H，而商在 AL 中是 05H。

```

MOVZX AX, BYTE1   ; 字节/字节
DIV  BYTE2        ; 余数:商在 AH:AL 中

```

在例 3 中，在 DX 中的余数是 1000H，而在 AX 中的商是 0080H。

```

MOV  DX, WORD2    ; 双字/字
MOV  AX, WORD3    ; 被除数在 DX:AX 中
DIV  WORD1        ; 余数:商在 DX:AX 中

```

例 4 首先把 WORD1 扩展成在 DX 中的双字。相除以后，在 DX 中的余数是 0000H，在 AX 中的商是 0002H：

```

MOV  AX, WORD1    ; 字/字
SUB  DX, DX       ; 被除数扩展到 DX 中
DIV  WORD3        ; 余数:商在 DX:AX 中

```

12.5.2 使用 IDIV 做带符号数除法

IDIV 指令是用于做带符号数据的除法。以下程序片断表示的是和上面给出的 4 个例子同样的例子，但用 IDIV 取代 DIV。

例 1 是 2000H(正数)除以 80H(负数)。在 AH 中的余数是 00H，在 AL 中的商是 C0H(-64)(使用同样的数据，DIV 产生+64 的商)：

```

MOV  AX, WORD1    ; 字/字节
IDIV BYTE1        ; 余数:商在 AH:AL 中

```

在例 2 中，商是 FB(-5)而余数是 EEH(-18)：

```

MOVZX AX, BYTE1   ; 字节/字节
IDIV  BYTE2        ; 余数:商在 AH:AL 中

```

在例 3 中, 商是 0080H(128), 而余数是 1000H(4 096):

```
MOV DX, WORD2      ; 双字/字
MOV AX, WORD3       ; 被除数在 DX:AX 中
IDIV WORD1          ; 余数: 商在 DX:AX 中
```

在例 4 中, 商是 0002H, 而余数是 0000H:

```
MOV AX, WORD1       ; 字/字
CWD                 ; 被除数扩展到 DX 中
IDIV WORD3          ; 余数: 商在 DX:AX 中
```

例 3 与例 4 和用 DIV 做时的答案是一样的。实际上, 如果被除数与除数的符号位相同, 那么 DIV 和 IDIV 就会产生同样的结果。但是, 如果被除数与除数的符号位不同, 则 DIV 产生一个正的商, 而 IDIV 则产生一个负的商。

下面的例子说明四字除以双字的情况。商是 DC5CH(56412):

```
DBLWD1 DD 0
DBLWD2 DD 225648
DBLWD3 DD 4
...
MOV EDX, DBLWD1     ; 四字/双字
MOV EAX, DBLWD2     ; 被除数在 EDX:EAX 中
IDIV DBLWD3         ; 余数: 商在 EDX:EAX 中
```

试用调试程序跟踪这些例子。

12.5.3 溢出与中断

DIV 和 IDIV 操作都假定商比原先的被除数小才是有意义的。因此, 操作可能容易产生溢出, 这时, 会发生中断, 产生不可预知的结果。例如, 除以 0 会引起中断, 但是除以 1 会产生一个与被除数相同的商, 也会引起一次中断。

这是一个有用的规则: 如果除数是一个字节, 那么它的值必须大于被除数左边的字节(AH); 如果除数是一个字, 那么它的值必须大于被除数左边的字(DX); 如果除数是一个双字, 它的值就必须大于被除数左边的双字(EDX)。

下面的指令用 1 作为除数, 虽然其他值可能也是适用的:

除法操作	被除数	除数	商
字除以字节:	012A	01	(1)2A
双字除以字:	0001 402B	0001	(1)402B
四字除以双字:	00000002 10542EB4	00000001	(2)10542EB4

在每种情况下, 商都超出了它的可用空间。通常的做法是在 DIV 或 IDIV 操作之前做一次测试, 如以下 2 个例子所示。在例 1 中, DIVRBYTE 是 1 字节的除数, 被除数在 AX 中:

```
CMP AH, DIVRBYTE    ; AH 和除数(字节)相比较
JNB L20             ; 如不小于, 则绕过
DIV DIVRBYTE        ; 字除以字节
```

在例 2 中, DIVRDWD 是双字的除数, 而被除数是在 EDX: EAX 中:

```

CMP  EDX, DIVRDWD      ; EDX 和除数(双字)相比较
JNB  L30                ; 如不小于, 则绕过
DIV  DIVRDWD            ; 四字除以双字

```

对于 IDIV, 逻辑上必须说明一个事实, 就是被除数或除数可能是负的。由于除数的绝对值必须是两个当中较小的一个, 所以可以使用 NEG 指令把一个负值临时设置成正的, 并在相除之后再恢复其符号。

12.5.4 用减法做除法

对于一个除数而言, 如果商太大了, 可以使用连续减的方法实现除法。也就是说, 从被除数中减去除数, 商的值加 1, 并且继续做减法直到被除数小于除数为止。在下面的例子中, 被除数在 AX 中, 除数在 BX 中, 而商则扩展到 CX 中:

```

      SUB  CX, CX          ; 清除商
L20:  CMP  AX, BX          ; 如果被除数 < 除数,
      JB   L30             ; 退出
      SUB  AX, BX          ; 从被除数中减去除数
      INC  CX              ; 商加 1
      JMP  L20             ; 重复
L30:  ...                  ; 商在 CX 中, 余数在 AX 中

```

在例行程序结束时, CX 的内容是商, 而 AX 的内容是余数。如果被除数在 DX:AX 中, 那么例行程序看来像是这样:

```

      SUB  CX, CX          ; 清除商
L20:  CMP  DX, 0            ; 如果 DX=0,
      JNE  L30             ; 绕过
      CMP  AX, BX          ; 如果被除数 < 除数,
      JB   L40             ; 退出
L30:  SUB  AX, BX          ; 从被除数中减去除数
      SBB  DX, 0            ; 减去进位标志
      INC  CX              ; 商加 1
      JMP  L20             ; 重复
L40:  ...                  ; 商在 CX 中, 余数在 AX 中

```

注意, 一个非常大的商和一个小的除数可能造成以牺牲处理时间为代价的数千次循环。

12.5.5 用移位做除法

对于除以 2 的幂(2, 4, 8 等等)的情况, 右移所要求的位数可以加快处理速度。以下不相关的例子假定被除数在 BX 中, 而 CL 含有 2:

```

右移 1 次:    SHR  BX, 01      ; 除以 2
右移 2 次:    SHR  BX, CL      ; 除以 4
右移 3 次:    SHR  BX, 03      ; 除以 8

```

下面的例行程序可以用于右移在 DX:AX 中的双字值。虽然指定的是移 4 位, 但可以适

用于其他值:

```
SHR AX, 04      ; AX 右移 4 位
MOV BL, DL      ; 把 DL 存入 BL
SHR DX, 04      ; DX 右移 4 位
SHL BL, 04      ; BL 左移 4 位
OR AH, BL       ; 把来自 BL 的 4 位插入 AH
```

12.5.6 符号变反

NEG(负)指令将二进制值的符号变反, 由正的变成负的, 反之亦然。实际上, NEG 将各位求反, 就像 NOT 一样, 然后加 1, 成为二进制补码表示法。NEG 的格式是

[label:]	NEG	register/memory
-----------	-----	-----------------

这里是一些不相关的例子:

```
NEG CL          ; 字节
NEG BX          ; 字
NEG EDI         ; 双字
NEG BINVAL      ; 存储器中的数据项
```

在 DX: AX 中的 32 位值的符号变反包括较多的步骤。NEG 不能同时对 DX: AX 起作用, 而对于 2 个寄存器都使用 NEG 会对它们都加 1, 这是无效的。取而代之的, 是使用 NOT 把各位变反, 并且使用 ADD 和 ADC 去加 1, 成为二进制补码:

```
NOT DX          ; 在 DX 中的位反向
NOT AX          ; 在 AX 中的位反向
ADD AX, 1       ; AX 加 1
ADC DX, 0       ; DX 加进位
```

剩下的一个次要的问题是: 按照程序本身定义的二进制数据或在一个外部文件中已经采用二进制格式的数据, 来实现算术运算是一件非常好的事。但是, 从键盘输入到程序中的数据都是按 ASCII 格式的。尽管 ASCII 数据适合于显示和打印, 但对于算术运算, 还需要专门地调整——这是下一章要讨论的课题。

12.6 数值数据处理器

这一节提供的是对数值处理器的一般介绍, 完整地讨论已经超出了本书的范围。Intel 数值数据处理器, 或协处理器为了实现取幂(乘方)、对数与三角这样一些操作而有其自己的指令系统和浮点硬件。8 个 80 位的浮点寄存器可以表示的数值能达到 10 的 400 次方, 并且执行起来要比正规的处理器快得多。

协处理器包含 8 个 80 位的寄存器 R1-R8, 采用下面的格式:

S	阶		有效数	
79	78	64	63	0

每个寄存器都有一个相关联的 2 位标记来指明它的状态：

00 包含一个有效数 10 包含一个无效数

01 包含一个零值 11 为空

协处理器识别 7 种类型的数值数据：

1. 字整数：16 位二进制数据。

S	数
15	14 0

2. 短整数：32 位二进制数据。

S	数
31	30 0

3. 长整数：64 位二进制数据。

S	数
63	62 0

4. 短实数：32 位浮点数据。

S	阶	有效数
31	30 23	22 0

5. 长实数：64 位浮点数据。

S	阶	有效数
63	62 52	51 0

6. 临时实数：80 位浮点数据。

S	阶	有效数
79	78 64	63 0

7. 压缩十进制数。18 个有效的十进制数字。

S	阶	有效数
79	78 72	71 0

类型 1、2 和 3 是通用二进制补码格式，类型 4、5 和 6 表示浮点数，类型 7 包含 18 个 4 位十进制数字。这些格式中的任何一种可以从存储器装入到协处理器的寄存器中，并且可以把寄存器的内容存入到存储器中。但是，关于它的计算，协处理器要把它在寄存器中的所有格式转换成临时实数。数据在存储器中的存放是按相反字节顺序的。

协处理器需要一个专门的操作把数值数据发送到协处理器，在那里完成操作并回送结果。

12.7 要 点

- 对于 1 字节累加器最大的带符号值是+127 和-128。
- 对于多字加法，使用 ADC 计算任何来自前面 ADD 的进位。如果操作是在一个循环中完成的，则使用 CLC 去清除进位标志。

- MUL 用于无符号数据, IMUL 用于带符号数据。
- 对于 MUL, 如果乘数是按字节定义的, 那么被乘数是 AL; 如果乘数是一个字, 则被乘数是 AX; 如果乘数是双字, 被乘数就是 EAX。
- 左移指令(SHL 或 SAL)可以用来乘以 2 的幂。
- DIV 用于无符号数据, IDIV 用于带符号数据。
- 对于除法, 如果除数被定义为字节, 则被除数就是 AX; 如果除数是字, 则被除数就是 DX: AX; 如果除数是双字, 则被除数是 EDX: EAX。
- 如果除数是一个字节, 那么该除数必须大于 AH 的内容; 如果除数是一个字, 就必须大于 DX 的内容; 或者如果除数是一个双字, 就必须大于 EDX 的内容。
- 右移指令可以用于除以 2 的幂的操作——SHR 用于无符号数, 而 SAR 用于带符号数。

12.8 习 题

12-1. 对于无符号和带符号数据, 它们的最大值是多少? (a) 一个字节, (b) 一个字, (c) 一个双字。

12-2. 按照算术操作的结果来区分进位与溢出。

12-3. 对于以下二进制加法, 表示出二进制的和以及其作为带符号与无符号十进制数的情况, 并给出溢出与进位标志的设置:

(a) 00010011	(b) 01010110	(c) 11010101	(d) 11011011
+ 00111000	+ 00111001	+ 01011010	+ 11010110
<hr/>			

12-4. “实现双字值的算术运算”一节包含 2 个把字对(第一个是 WORD1A)相加的程序片断。修改这 2 个例子, 使得它们用 3 个字对取代 2 个字对进行相加。定义附加的字为 WORD3A 和 WORD3B 并把老的 WORD3A 和 WORD3B 改成 WORD4A 和 WORD4B。

对于 12-5 题到 12-8 题, 引用以下数据, 按相反顺序正确定义字:

BIN_AMT1	DW	0147H
	DW	139AH
BIN_AMT2	DW	02B3H
	DW	2D41H
CAL_AMT	DW	0
	DW	0
	DW	0

12-5. 编写指令完成下列相加操作: (a) 字 BIN_AMT1 加到字 BIN_AMT2, (b) 起始于 BIN_AMT1 的双字加到 BIN_AMT2 处的双字。

12-6. 解释以下相关指令的作用:

```
STC
MOV  BX, BIN_AMT1
ADC  BX, BIN_AMT2
```

12-7. 编写指令做以下乘法(MUL): (a) 字 BIN_AMT1 乘以字 BIN_AMT2, (b) 起始于

BIN_AMT1 的双字乘以字 BIN_AMT2。乘积存入 CAL_AMT。

12-8. 编写指令做以下除法(DIV)：(a)字 BIN_AMT1 除以 24H，(b)起始于 BIN_AMT1 的双字除以字 BIN_AMT2。商存入 CAL_AMT。

12-9. 除零以外的什么除数会造成溢出错误？

12-10. 参考“用移位做乘法”一节中把 DX:AX 对左移 4 位的例子。修改该例子为左移 2 位。

12-11. 参考“用移位做除法”中把 DX:AX 对右移 4 位的例子。修改该例子为右移 2 位。

算术运算 II :

处理 ASCII 和 BCD 数据

目的：研究 ASCII 与 BCD 数据格式，实现这些格式的算术运算，以及这些格式与二进制格式之间的转换。

13.1 引言

在计算机上进行算术运算的自然数据格式是二进制。正如我们在第 12 章所见到的那样，只要是程序本身定义的数据，二进制格式是不会发生什么大问题的。但是，程序必须处理的大量数据，并不都是采用二进制格式。例如，从键盘输入程序的数值数据是以 10 为基数的 ASCII 字符格式。类似地，在屏幕上显示的数字值也是 ASCII 格式。

一种相近的数值格式——二进制编码的十进制数，又称二-十进制数会偶然用到，并且以非压缩与压缩的形式出现。PC 提供许多便于简单算术运算与两种格式之间转换的指令。本章也会涉及有关把 ASCII 数据转换成二进制格式以便完成算术运算的技术，以及为了观察而把二进制结果反过来转换成 ASCII 格式的技术。在本章末尾的程序中，综合了第 1 章到第 12 章所涉及的许多资料。

在如 C 那样的高级语言中，编译程序是要解决小数点(十进制或二进制的)问题的。但是，计算机和汇编程序都不能识别在一个运算字段中的小数点，所以汇编语言程序员就必须说明它的位置。

本章所介绍的指令是：

AAA	加法后的 ASCII 调整	AAD	除法的 ASCII 调整
AAS	减法后的 ASCII 调整	DAA	加法后的十进制调整
AAM	乘法后的 ASCII 调整	DAS	减法后的十进制调整

13.2 十进制格式的数据

到现在为止，程序举例都是以二进制与 ASCII 格式来处理数字值的。处理器还支持二进制(BCD)格式，它允许进行一些受限制的算术操作。BCD 格式的两种用法是：

1. 允许不损失精度的数的正常舍入，这一特性对于处理美元和美分时特别有用(表示美

元和美分的二进制数的舍入可能造成精度的损失)。

2. 对于完成从键盘输入的小值的算术运算，或者对于在屏幕或打印机上的输出来说，BCD 是个比较简单的格式。

BCD 数字由 4 位组成，表示十进制数字 0 到 9：

二进制	BCD 数字	二进制	BCD 数字
0000	0	0101	5
0001	1	0110	6
0010	2	0111	7
0011	3	1000	8
0100	4	1001	9

可以按非压缩的或压缩的形式存放 BCD 数字：

(1) 非压缩的 BCD 包含在每个字节较低(最右边)4 位中的单个 BCD 数位，较高的 4 位是 0。注意，虽然 ASCII 格式的数据在某种意义上也是“非压缩的”，但是不那样叫它。

(2) 压缩的 BCD 包含 2 个 BCD 数位：一个是较高的 4 位，一个是较低的 4 位。这种格式通常用于使用数值协处理器的算术运算，由 DT 伪操作定义成 10 个字节。

下面是用 3 种格式表示十进制数 1527 的例子：

格式	长度	内容
ASCII	4 字节	31 35 32 37
非压缩的 BCD	4 字节	01 05 02 07
压缩的 BCD	2 字节	15 27

处理器每次一个数位地完成 ASCII 和 BCD 值的运算。必须使用专门的指令来做两种格式之间的转换。

13.3 处理 ASCII 数据

由于从键盘输入的数据是 ASCII 格式，所以输入的十进制值如 1234 在存储器中的表示是 31323334H。执行 ASCII 值算术运算的指令包括 AAA 和 AAS：

[label:]	AAA	；加法后的 ASCII 调整
[label:]	AAS	；减法后的 ASCII 调整

这些指令被编码成没有操作数并自动调整 AX 寄存器中的 ASCII 值。调整的结果是由 ASCII 值表示一个非压缩的基数为 10 的数，而处理器要执行的是基数为 2 的算术运算。

13.3.1 ASCII 数相加

研究下面 ASCII 数相加的 3 个例子的结果：

例 1 .	35H	例 2 .	38H	例 3 .	39H
	+ 32H		+ 34H		+ 39H
和	67H		6CH		72H

在例 1 中, 尽管最左边的数位是需要校正的, 但最右边的数位的 7 是正确的和。在例 2 中由于进位由十进制变成了十六进制数位, 因此 6CH 不是正确的 ASCII 和。由于进位是从个位到十位位置, 所以例 3 是不正确的。

为了校正一个 ASCII 和, AAA 操作检查 AL 寄存器最右边的十六进制数位。如果数位在 A 与 F 之间或辅助进位标志(AF)是 1, 则该操作就把 6 加到 AL 上, 把 1 加到 AH 上, 并且把进位标志(CF)和辅助进位标志都置成 1。在所有情况下, AAA 都要把 AL 最左边的十六进制数位清除为零。(为什么加 6? 因为那是十六进制(16)和十进制(10)之间的差)。

让我们看看 AAA 是如何处理前面的 3 个例子的。假设 ASCII 数是在 AL 和 BL 中, 指令是:

```
ADD AL, BL      ; 加 ASCII 数
AAA             ; 对 ASCII 加进行调整
```

例 1. 35H 与 32H 的和是 67H。AAA 检查最右边的数位(7)。因为它不在 A 与 F 之间, 也没有设置 AF 标志, 所以 AAA 简单地把最左边的数位(6)清除为 0。

例 2. 38H 与 34H 的和是 6CH。由于最右边的数位(C)是在 A 与 F 之间, AAA 执行以下操作:

	AH	AL
	00	6C
把 6 加到 AL	00	72
把 1 加到 AH	01	72
清除 AL 最左边的数位	01	02 (和=12)

例 3. 39H 与 39H 的和是 72H。虽然最右边的数位(2)不是在 A 与 F 之间, 但由于进位是进到十位位置, 所以 AF 标志被置成 1。AAA 完成以下操作:

	AH	AL
	00	72
把 6 加到 AL	00	78
把 1 加到 AH	01	78
清除 AL 最左边的数位	01	08 (和=18)

这些和 0006、0102 和 0108 是技术上的 BCD 数。为了恢复 ASCII 表示, 在 AH 与 AL 的最左边的十六进制数位中简单地插入 3:

```
OR AX, 3030H      ; 把 BCD 转换成 ASCII
```

对于 1 个字节的 ASCII 数相加来说, 这一切都是非常好的。但是, 多字节的 ASCII 数相加就需要一个循环, 该循环从右到左(低阶位到高阶位)进行处理, 并把进位计算在内。图 13-1 中的部分程序是把 2 个 3 字节的 ASCII 数(ASCVALUE1 和 ASCVALUE2)相加, 并产生 4 字节的和 ASCTOTAL。注意以下几点:

- 起点的 CLC 指令把 CF 标志置成 0。
- 其后的 A20 处, MOVZX 指令把相继的 ASCII 字符装入 AL。由于其后的 AAA 可能要向 AH 加 1, 所以 MOVZX 还要清除 AH, 它会存在于下一个循环里。注意, 使用 XOR 或 SUB 来清除 AH 会改变 CF 标志。
- ADC 用于做加法, 因为它能自动地把任何进位从 AL 加到 AH。

- MOV 按 ASCTOTAL 的相继字节存放每一个 ASCII “和”。
- 当循环完成时,程序把 AH(包含最后的 00 或 01)传送到 ASCTOTAL 的最左边的字节。

最后, ASCTOTAL 是 01020702H。为了在每个字节中插入 ASCII 3, 程序步进地通过 ASCTOTAL, 并用 30H 去 OR(或)每个字节。结果是 31323732H, 即十进制的 1272, 在结束之前程序会把它显示出来。

ASCVALUE1	DB	'548'	; ASCII 项
ASCVALUE2	DB	'724'	
ASCTOTAL	DB	'0000'	
.386		...	
		CLC	; 加 ASCII 值:
		LEA	SI, ASCVALUE1+2 ; 初始化 ASCII
		LEA	DI, ASCVALUE2+2 ; 数的地址
		LEA	BX, ASCTOTAL+3
		MOV	CX, 03 ; 初始化 3 次循环
A20:			
		MOVZX	AX, [SI] ; ASCII 字节装入 AX
		ADC	AL, [DI] ; 加 (带进位)
		AAA	; ASCII 调整
		MOV	[BX], AL ; 存放和
		DEC	SI
		DEC	DI
		DEC	BX
		LOOP	A20 ; 循环 3 次
		MOV	[BX], AH ; 最后, 存进位
		LEA	BX, ASCTOTAL+3 ; 把 ASCTOTAL
		MOV	CX, 04 ; 转换成 ASCII 格式
A30:			
		OR	BYTE PTR [BX], 30H
		DEC	BX
		LOOP	A30 ; 循环 4 次
		MOV	AX, 1300H ; 请求显示
		MOV	BX, 0031H ; 页与属性
		LEA	BP, ASCTOTAL ; ASCII 行
		MOV	CX, 04 ; 行的长度
		MOV	DX, 0824H ; 行与列
		INT	10H
		...	

图 13-1 ASCII 数相加

程序在 AAA 之后没有用 OR 去插入最左边的 3, 因为 OR 要设置进位标志并会改变 ADC 指令的结果。一个解决方案是通过标志寄存器进栈(PUSH)来保存标志的设置, 执行 OR, 然后把标志出栈(POPF)以恢复标志的设置:

```

ADC AL, [DI]      ; 带进位加
AAA              ; 对 ASCII 进行调整
PUSHF            ; 保存标志
OR AL, 30H       ; 插入 ASCII 3
POPF            ; 恢复标志
MOV [BX], AL     ; 存放和

```

13.3.2 ASCII 数相减

AAS 指令的工作非常像 AAA。AAS 检查 AL 的最右边的十六进制数位(4 位)。如果该数位在 A 到 F 之间或辅助进位为 1, 则该操作从 AL 中减去 6, 从 AH 中减去 1, 并且设置辅助

进位标志(AF)和进位标志(CF)为 1。在任何情况下, AAS 都要清除 AL 最左边的十六进制数位。

以下两个例子假定 ASCVALUE1 是 39H, ASCVALUE2 是 35H。第一个例子是从 ASCVALUE1(39H)中减去 ASCVALUE2(35H)。AAS 不需要进行调整, 因为最右边的十六进制数位是小于十六进制 A 的:

	AX	AF	CF
MOV AL, ASCVALUE1	: 0039		
SUB AL, ASCVALUE2	: 0004	0	0
AAS	: 0004	0	0
OR AL, 30H	: 0034		

第二个例子是从 ASCVALUE2(35H)中减去 ASCVALUE1(39H)。由于结果的最右边数位是十六进制的 C, 所以 AAS 从 AL 中减去 6, 从 AH 中减去 1, 并设置 AF 和 CF 标志:

	AX	AF	CF
MOV AL, ASCVALUE2	: 0035		
SUB AL, ASCVALUE1	: 00FC	1	1
AAS	: FF06	1	1

答案应该是-4, 即 FF06H, 它是个十的补码, 即十进制 $-10+6=-4$ 。

13.3.3 ASCII 数相乘

ASCII 数的乘法和除法首先需要把它们转换成非压缩的 BCD 格式, 然后可以使用 AAM 和 AAD 指令直接地去执行非压缩的 BCD 数的算术运算:

[label:]	AAM	; 乘法之后的 ASCII 调整
[label:]	AAD	; 除法之前的 ASCII 调整

AAM 指令校正正在 AX 寄存器中相乘的 ASCII 数据的结果。但是, 必须首先清除每个字节最左边的十六进制数位 3, 从而把该值转换成非压缩的 BCD, 使 AAM 实际地校正 BCD 数据(不是 ASCII 数据)。例如, ASCII 数 31323334H 变成 01020304H 成为非压缩的 BCD。还有, 由于每次只是调整一个字节, 所以可以只乘 1 字节字段并必须在一个循环中重复执行该操作。只使用 MUL(无符号乘法)操作(不是 IMUL 操作)。

AAM 把 AL 除以 10(0AH)并把商存放到 AH, 把余数存放到 AL。例如, 假设 AL 包含 ASCII 35H, CL 包含 39H。以下的代码是把 AL 的内容乘以 CL 并把结果转换成 ASCII 格式:

指令	注释	AX	CL
...	; 初始值	0035	39
AND CL, 0FH	; 把 CL 转换成 09	0035	09
AND AL, 0FH	; 把 AL 转换成 05	0005	09
MUL CL	; AL 乘以 CL	002D	09
AAM	; 把 AX 转换成非压缩的 BCD	0405	
OR AX, 3030H	; 把 AX 转换成 ASCII	3435	

MUL 操作在 AX 中产生 002DH(45)。AAM 把这个值除以 0AH，产生的商 04 在 AH，而余数 05 在 AL。然后 OR 指令把非压缩的 BCD 值转换成 ASCII 格式。

图 13-2 的部分程序描述的是 4 字节 ASCII 被乘数与 1 字节 ASCII 乘数的相乘。因为 AAM 只能适于 1 字节操作，所以程序要每次一个字节地步进地从右到左经过被乘数。最后，非压缩的 BCD 乘积是 0108090105，在循环到 A30 处转换成正确的 ASCII 格式 3138393135，即十进制的 18915。在结束处理之前，程序显示这个乘积。

如果乘数大于一个字节，那么你必须还要提供另外一个循环，该循环步进地经过乘数。在那种情况下，把 ASCII 数据转换成二进制格式可能更为简单。下一节会涉及这一问题。

	MULTCAND	DB	'3783'	; ASCII 项
	MULTPLER	DB	'5'	
	ASCPROD	DB	5 DUP(0)	
	...			
	MOV	CX	04	; 初始化 4 次循环
	LEA	SI	MULTCAND+3	
	LEA	DI	ASCPROD+4	
	AND		MULTPLER, 0FH	; 清除 ASCII 3
A20:	MOV	AL	[SI]	; 装入 ASCII 字符
	AND	AL	0FH	; 清除 ASCII 3
	MUL		MULTPLER	; 乘
	AAM			; ASCII 调整
	ADD	AL	[DI]	; 加到
	AAA			; 所存放的
	MOV		[DI], AL	; 乘积上
	DEC		DI	
	MOV		[DI], AH	; 存放乘积的进位
	DEC		SI	
	LOOP		A20	; 循环 4 次
	LEA	BX	ASCPROD+4	; 把乘积转换成 ASCII:
	MOV		CX, 05	; 从右到左, 5 个字节
A30:	OR		BYTE PTR [BX], 30H	
	DEC		BX	
	LOOP		A30	; 循环 5 次
	MOV	AX	1300H	; 请求显示
	MOV	BX	0031H	; 页与属性
	LEA	BP	ASCPROD	; ASCII 行
	MOV		CX, 05	; 行的长度
	MOV	DX	0824H	; 行与列
	INT		10H	
	...			

图 13-2 ASCII 数相乘

13.3.4 ASCII 数相除

AAD 指令在除法之前提供对于 ASCII 被除数的校正。正如 AAM 一样，首先要从 ASCII 字节中清除最左边的一些 3，产生非压缩的 BCD 格式。AAD 允许 2 字节的被除数放在 AX 中。除数可以是只有单字节的 01 到 09。

假定 AX 是 ASCII 值 28(3238H)，CL 是除数 ASCII 7(37H)。以下各指令实现调整与除法：

指令	注释	AX	CL
...	; 初始值	3238	37
AND CL, 0FH	; 转换成非压缩的 BCD	3238	07
AND AX, 0F0FH	; 转换成非压缩的 BCD	0208	
AAD	; 转换成二进制	001C	
DIV CL	; 除以 7	0004	

AAD 使 AH 乘以 10(0AH)，把乘积 20(14H) 加到 AL 上，并且清除 AH。结果 001CH 是十进制 28 的十六进制表示。

图 13-3 中的部分程序是 1 字节的除数去除 4 字节被除数的除法。程序从左到右步进地经过被除数。LODSB 从 DIVIDEND 中取一个字节到 AL(通过 SI)中，而 STOSB 把来自 AL 的字节存入 ASCQUOT(通过 DI)。余数仍保留在 AH 中，使得 AAD 能在 AL 中调整它。最后，非压缩的 BCD 格式的商是 00090204，而在 AH 中的余数是 02。程序把商转换成 ASCII 格式的 30393234(这次是从左到右)，并显示 ASCII 商 0924。

如果除数大于一个字节，那么你就必须还要提供另一个循环，步进地经过除数。还有更好的办法，见后面的“ASCII 数据转换成二进制格式”一节。

	DIVIDEND	DB	'3698'	; ASCII 项
	DIVISOR	DB	'4'	
	ASCQUOT	DB	4 DUP(0), '\$'	
	...			
		MOV	CX, 04	; ASCII 数相除:
		SUB	AH, AH	; 初始化 4 次循环
		AND	DIVISOR, 0FH	; 清除被除数的左边字节
		LEA	SI, DIVIDEND	; 清除 ASCII 3 的除数
		LEA	DI, ASCQUOT	
A20:		LODSB		; 装入 ASCII 字节
		AND	AL, 0FH	; 清除 ASCII 3
		AAD		; 除法调整
		DIV	DIVISOR	; 除
		STOSB		; 存商
		LOOP	A20	; 循环 4 次
				; 把商转换成 ASCII:
		LEA	BX, ASCQUOT	; 最左边字节
		MOV	CX, 04	; 4 个字节
A30:		OR	BYTE PTR[BX], 30H	; 清除 ASCII 3
		INC	BX	; 下一个字节
		LOOP	A30	; 循环 4 次
		MOV	AH, 09H	; 显示
		LEA	DX, ASCQUOT	; 商
		INT	21H	
		...		

图 13-3 ASCII 数相除

13.4 处理压缩的 BCD 数据

在前面 ASCII 除法的例子中，商是 00090204。如果压缩这个值，只保留每个字节右边的数位，结果就成了 0924，而现在是在压缩的 BCD 格式。可以使用 2 条十进制调整指令 DAA 和 DAS 对压缩的 BCD 数据实现加法和减法：

[label:]	DAA	; 加法后的十进制调整
[label:]	DAS	; 减法后的十进制调整

DAA 校正在 AL 中的 2 个压缩的 BCD 值相加的结果, 而 DAS 校正它们相减的结果。再次提醒: 你必须每次一个字节(两个数位)地处理 BCD 字段。

在 AL 中的 BCD 和是由 2 个 4 位的数位组成的。如果最右边数位的值超过 9 或 AF 标志被置 1, 那么 DAA 向 AL 加 6 并把 AF 置 1。如果现在 AL 中的值超过 99H 或 CF 被置 1, 那么 DAA 向 AL 加 60H 并把 CF 置 1。否则, 它清除 AF 和 CF。以下的例子将说明这一过程。

考虑 BCD 值 057836 和 069427 相加。由于 CF 标志被清除为 0, 所以开始把在 AL 中的最右边一对数位相加, 即 36+27:

	BCD	十六进制	二进制	被存放的 BCD
第一个 ADC, 清除 CF	36	36	0011 0110	
	<u>27</u>	<u>27</u>	<u>0010 0111</u>	
	63	5D	0101 1101	
DAA 加 06H, AF 置!			<u>0000 0110</u>	
			<u>0110 0011</u>	63
第二个 ADC, CF 置!	78	78	0111 1000	
	<u>94</u>	<u>94</u>	<u>1001 0100</u>	
	(1) 72	(1) 0C	(1) 0000 1100	
DAA 加 06H, AF 置!			<u>0000 0110</u>	
			<u>0001 0010</u>	
DAA 加 60H (因为 CF 为 1), CF 置!			<u>0110 0000</u>	
			<u>0111 0010</u>	72
第三个 ADC, 加!	05	05	0000 0101	
(因为 CF 为 1),	06	06	0000 0110	
清除 CF	<u>1</u>	<u>1</u>	<u>0000 0001</u>	
	12	0C	0000 1100	
DAA 加 06H, AF 置!			<u>0000 0110</u>	
			<u>0001 0010</u>	12

现在, BCD 的和正确地存放为 127263。

图 13-4 中的部分程序说明上述 BCD 加法的例子。过程 B10CONVRT 把 ASCII 值 ASCVALUE1 和 ASCVALUE2 分别转换成压缩的 BCD 值 BCDVALUE1 和 BCDVALUE2。从右到左进行的处理可以和从左到右一样的容易。另外, 处理字比处理字节更容易, 这是因为需要 2 个 ASCII 字节去产生一个压缩的 BCD 字节。但是, 使用字要求 ASCII 字段的字节数为偶数。

该程序完成 3 次循环把压缩的 BCD 数加到 BCDSUM 上。最后的和是 00127263H, 它是可以用 DEBUG 来核定的。把 BCD 和转换成 ASCII 并显示它是一个有用的练习。

ASCVALUE1	DB	'057836'	; ASCII 数据项
ASCVALUE2	DB	'069427'	
BCDVALUE1	DB	'000'	; BCD 数据项
BCDVALUE2	DB	'000'	
BCDSUM	DB	4 DUP(0)	
.386			
	...		
	LEA	SI, ASCVALUE1+4	; 初始化 ASCII
	LEA	DI, BCDVALUE1+2	; 和 BCD 值
	CALL	B10CONVRT	; 调用转换例行程序
	LEA	SI, ASCVALUE2+4	; 初始化 ASCII
	LEA	DI, BCDVALUE2+2	; 和 BCD 值
	CALL	B10CONVRT	; 调用转换例行程序
	XOR	AH, AH	; 加 BCD 数:
			; 清除 AH
	LEA	SI, BCDVALUE1+2	; 初始化
	LEA	DI, BCDVALUE2+2	; BCD
	LEA	BX, BCDSUM+3	; 地址
	MOV	CX, 03	; 3 字节字段
	CLC		
A20:			
	MOV	AL, [SI]	; 取 BCDVALUE1
	ADC	AL, [DI]	; 加 BCDVALUE2
	DAA		; 十进制调整
	MOV	[BX], AL	; 存入 BCDSUM
	DEC	SI	
	DEC	DI	
	DEC	BX	
	LOOP	A20	; 循环 3 次
	...		
		Convert ASCII to BCD:	

B10CONVRT	PROC		
	MOV	CX, 03	; 字转换
B20:	MOV	AX, [SI]	; 取 ASCII 对
	XCHG	AH, AL	
	SHL	AL, 04	; 移掉
	SHL	AX, 04	; ASCII 3
	MOV	[DI], AH	; 存 BCD 数字
	DEC	SI	
	DEC	SI	
	DEC	DI	
	LOOP	B20	; 循环 3 次
	RET		
B10CONVRT	ENDP		

图 13-4 BCD 数转换与相加

13.5 ASCII 数据转换成二进制格式

按 ASCII 或 BCD 格式完成算术运算只适于短字段。对于大多数算术运算来说, 把这样的数转换成二进制格式将更为实用。实际上, 从 ASCII 直接转换成二进制比从 ASCII 转换到 BCD 再转换成二进制要更加容易。

从 ASCII 格式转换成二进制是基于这样的事实: ASCII 数是以 10 为基数的, 而计算机执行算术运算是以 2 为基数的。下面是这一过程:

1. 由 ASCII 字段的最右边字节开始, 并从右到左进行处理。
2. 从每个 ASCII 字节左边的十六进制数位中去掉 3, 形成一个压缩的 BCD 数。
3. 第一个(最右边的)BCD 数位乘以 1, 第二个数位乘以 10(0AH), 第三个乘以 100(64H),

以此类推，并计算乘积的总和。

以下例子是把 ASCII 数 3569 从右到左转换成二进制数：

十进制		十六进制	
步骤	乘积	步骤	乘积
9×1=	9	9×01H=	9H
6×10=	60	6×0AH=	3CH
5×100=	500	5×64H=	1F4H
3×1000=	3000	3×3E8H=	668H
和:	3569		0DF1H

试检查一下和 0DF1H，实际上它等于十进制的 3569。

在下一例子中，部分程序把 ASCII 数 3569 转换成它的二进制等效值：

```

ASCLENTH    EQU    4                ; ASCII 长度
ASCVALUE    DB     '3569'          ; ASCII 值
BINVALUE    DW     0                ; 二进制和
MULTFACT    DW     1                ; 1, 10, 100, ...
...
MOV CX, ASCLENTH                    ; 循环计数值
LEA SI, ASCVALUE+3                  ; ASCVALUE 的地址

L10:
MOV AL, [SI]                        ; 选择 ASCII 字符
AND AX, 000FH                       ; 去掉 3 的区段
MUL MULTFACT                         ; 乘以 10 的因子
ADD BINVALUE, AX                    ; 加到二进制数
MOV AX, MULTFACT                    ; 计算下一个
IMUL AX, 10                          ; 10 的因子
MOV MULTFACT, AX
DEC SI                               ; 是最后的 ASCII 字符吗?
LOOP L10                             ; 否, 继续

```

LEA 指令初始化 ASCII 字段的最右边字节的地址，把 ASCVALUE+3 放在 SI 中。在 L10 处的指令把 ASCII 字节传送到 AL，即 MOV AL, [SI]。该操作使用 ASCVALUE+3 的地址把 ASCVALUE 最右边的字节复制到 AL 中。循环的每次迭代都使 SI 减 1，并访问左边的下一个字节。循环对于 ASCVALUE 的 4 个字节中每一个进行重复。还有，每次迭代 MULTFACT 乘以 10(0AH)，给出的乘数是 1, 10, 100，以及 1000。最后，BINVALUE 包含正确的二进制值 F10DH，是以相反字节顺序排列的。

在下一节中的程序是反过来把二进制值转换成十进制格式。

13.6 二进制数据转换成 ASCII 格式

为了打印或显示二进制算术运算的结果，首先必须把它转换成 ASCII 格式。该操作是把前述的步骤反过来进行：取代乘法的是重复地用 10(0AH)去除二进制数，直到商小于 10。每个余数(只能是 0 到 9)相继地产生 ASCII 数。作为一个例子，让我们把 0DF1H 反过来转换成十进制格式：

除以 0AH	商	余数
DF1 ÷ A	164	9
164 ÷ A	23	6
23 ÷ A	3	5

由于商(3)现在小于除数(0AH), 所以操作完成。从右到左的余数和最后的商一起形成 BCD 结果: 3569。剩下的所有事情是把这些数字与 ASCII 3 一起存入存储器中, 成为 33353639。

以下的例子是把二进制数 0DF1H 转换为 ASCII 格式:

```

ASCVALUE    DB  4 DUP( ' ' )           ; 数据项
BINVALUE    DW  0DF1H
...
MOV  CX, 0010                               ; 除法因子
LEA  SI, ASCVALUE+3                         ; ASCVALUE 的地址
MOV  AX, BINVALUE                           ; 得到二进制值数

L20:
CMP  AX, CX                               ; 值<10?
JB   L30                                   ; 是, 退出
XOR  DX, DX                               ; 清除高位商
DIV  CX                                   ; 除以 10
OR   DL, 30H
MOV  [SI], DL                             ; 存 ASCII 字符
DEC  SI
JMP  L20

L30:
OR   AL, 30H                               ; 存最后的商
MOV  [SI], AL                             ; 作为 ASCII 字符

```

该例子是把二进制数相继地除以 10(0AH), 直到剩下的商小于 10 为止, 并以 ASCII 格式存放所产生的十六进制数位为 33353639。你可能发现复制这个程序并步进地跟踪它的执行是有用的。

13.7 乘积的移位与舍入

假设一个乘积有 3 个十进制小数位, 而必须舍入它, 并且使它减少到 2 个十进制小数位。例如, 如果乘积是 17.385, 在最右边的(不希望有的)十进制小数位置上加 5, 并右移一个数位:

```

乘积:      17.385
加 5:      + 0.005
舍入的乘积: 17.390 =17.39

```

如果(a)乘积是 17.3855, 加 50 并移 2 个数位, 并且如果(b)乘积是 17.38555, 加 500 并移 3 个数位:

<pre> (a) 17.3855 + 0.0050 ----- 17.3905 =17.39 </pre>	<pre> (b) 17.38555 + 0.00500 ----- 17.39055 =17.39 </pre>
--	---

进一步说，有 6 个十进制小数位的数需要加 5000 并且要移 4 个数位，以此类推。现在，由于计算机通常都是处理二进制数据，17.385 表示为 43E9H。43E9H 加 5 得到 43EEH 或十进制格式的 17390。到此为止，一切良好。但是，移一个二进制数位结果形成 21F7H 或 8695——确实，简单的移位使该值减半。你需要等效于右移一个十进制数位的移位。可以用把被舍入的二进制值除以 10(或 hex A)的办法完成这种移位：hex 43EE 除以 hex A=6CBH。把 6CBH 转换成十进制数得到 1739。现在，在正确的位置上正好插入十进制的小数点，并且你可以显示这个被舍入的移位的值为 17.39。

用这一方法，可以舍入与移位任何二进制数。对于 3 个十进制小数位的，是加 5 并除以 10；对于 4 个十进制小数位的，是加 50 再除以 100。也许你已经注意到了一种模式：舍入因子(5, 50, 500 等等)总是移位因子值(10, 100, 1000 等等)的一半。

当然，二进制数中的小数点是隐含的，并且实际上是不出现的。

程序：ASCII 与二进制数据之间的转换

在图 13-5 中的程序允许用户输入量和速率，并显示所计算的值。例如，量可以是千瓦小时或加仑。为简洁起见，程序省略了某些出错检查，不然这些检查是应该包括在内的。另外，每个过程都指明了所用的寄存器，而不是让它们进栈和出栈。这些过程如下：

- A10MAIN 处理初始化并为输入数据与计算值调用过程。
- B10INPUT 从键盘接受 ASCII 格式的量 and 速率，这些值可以包含十进制的小数点。
- C10QTY 初始化从 ASCII 量到二进制的转换。
- D10RATE 初始化从 ASCII 速率到二进制的转换。
- E10MULT 实现乘法，并对具有 3 个或更多十进制小数位的任何乘积进行舍入与移位。
- F10PROD 插入十进制小数点，确定最右边的位置，以便开始存放 ASCII 字符，并且把二进制的乘积转换成 ASCII 值。

```

TITLE      A13CALC (EXE) 输入量和速率的 ASCII 值，确定十进制小数位数，
;          转换为二进制，计算乘积，显示 ASCII
;
.MODEL SMALL
.STACK 64
.DATA
LEFTCOL    EQU    28          ; 屏幕位置的
RIGHTCOL   EQU    52          ; 等价符号
TOPROW     EQU    10
BOTROW     EQU    14
QTYPARAM   LABEL BYTE
MAXQLEN    DB    6            ; 键盘输入的
ACTQLEN     DB    ?           ; 量参数表
QTYFLD     DB    6 DUP(?)     ;
RATEPAR    LABEL BYTE
MAXRLEN    DB    6            ; 键盘输入的
ACTRLEN     DB    ?           ; 速率参数表
RATEFLD     DB    6 DUP(?)    ;
PROMPT1    DB    'Quantity? '
PROMPT2    DB    'Rate? '
PROMPT3    DB    'Product = '
ASCPROD    DB    10 DUP(30H)
PROMPT4    DB    'Press any key to continue or Esc to quit'
ADJUST     DW    ?            ; 数据项

```

图 13-5 处理 ASCII 与二进制数据

```

BINPROD    DW    00
BINQTY     DW    00
BINRATE    DW    00
COL        DB    00
DECIND     DB    00
MULT10     DW    01
NODECIMS   DW    00
ROW        DB    00
SHIFT      DW    ?
TENWD      DW    10
.386 ;-----
.CODE
A10MAIN    PROC    FAR
MOV        AX,@data           ; 初始化 DS
MOV        DS,AX              ; 和ES寄存器
MOV        ES,AX
MOV        AX,0003H           ; 设置显示方式
INT        10H                 ; 和清除屏幕
A20:       CALL    Q10WINDOW    ; 清除窗口
CALL       B10INPUT            ; 接受量和速率
CALL       C10QTY              ; 把量转换为二进制
CALL       D10RATE             ; 把速率转换为二进制
CALL       E10MULT             ; 计算乘积,舍入
CALL       F10PROD             ; 把乘积转换成 ASCII
CALL       G10FORMAT           ; 显示乘积
CALL       H10PAUSE            ; 用户暂停
CMP        AL,1BH              ; 按Esc了吗?
JNE        A20                 ; 否,继续
MOV        AX,4C00H            ; 结束处理
INT        21H
A10MAIN    ENDP

;-----
; 从键盘接受量和速率:
;-----
B10INPUT    PROC    NEAR
MOV        ROW,TOPROW+1        ; 使用 AX, BP, CX, SI
MOV        COL,LEFTCOL+3
LEA        BP,PROMPT1          ; 量的提示符
MOV        CX,15                ; 字符数
CALL       K10DISPLY            ;
MOV        AH,0AH              ; 从键盘
LEA        DX,QTYPARAM          ; 接受量
INT        21H
MOV        COL,LEFTCOL+3        ; 设置列
INC        ROW                  ; 下一行
LEA        BP,PROMPT2          ; 速率提示符
MOV        CX,15                ; 字符数
CALL       K10DISPLY            ;
MOV        AH,0AH              ; 从键盘接受
LEA        DX,RATEPAR           ; 速率
INT        21H
INC        ROW                  ; 下一行
RET
B10INPUT    ENDP

;-----
; 把量转换为二进制:
;-----
C10QTY      PROC    NEAR
MOV        NODECIMS,00          ; 使用 AX, CX, SI
MOVZX      CX,ACTQLEN           ; 消除十进制小数位
LEA        SI,QTYFLD-1          ; 量的长度
ADD        SI,CX                ; 设置量的
                                ; 右进位置
CALL       J10ASCBIN           ; 转换成二进制
MOV        AX,BINPROD           ; 按二进制量
MOV        BINQTY,AX           ; 存放结果
RET
C10QTY      ENDP

;-----
; 把速率转换为二进制:
;-----
D10RATE     PROC    NEAR
MOVZX      CX,ACTRLEN           ; 使用 AX, CX, SI
LEA        SI,RATEFLD-1        ; 速率的长度
ADD        SI,CX                ; 设置速率的
                                ; 右进位置
CALL       J10ASCBIN           ; 转换成二进制
MOV        AX,BINPROD           ; 按二进制速率
MOV        BINRATE,AX          ; 存放结果

```

图 13-5 续

```

RET
D10RATE ENDP
;
;
; 量乘以速率, 舍入并移位, 乘积基于十进制小数位数:
; -----
E10MULT PROC NEAR ; 使用 AX, CX, SI
MOV CX, 10 ; 乘积的长度
LEA DI, ASCPROD ; 设置 ASCII 乘积
MOV AL, 30H ; 为 30
CLD
REP STOSB
MOV SHIFT, 10 ; 初始移位因子
MOV ADJUST, 00

MOV CX, NODECIMS
CMP CL, 06 ; 如果大于 6 个
JA E40 ; 十进制数, 出错
SUB CX, 02
JLE E30 ; 如果 0, 1, 2 个十进制数
MOV NODECIMS, 02 ; 则通过, 否则把十进制小数位置为 2
MOV AX, 01 ; 计算
E20: IMUL AX, 10 ; 移位因子
LOOP E20 ; = 1, 10, 100, ...
MOV SHIFT, AX
SHR AX, 1 ; 舍入值=移位的一
MOV ADJUST, AX ; 半 = 0, 5, 50, ...
E30: MOV AX, BINQTY ; 量 x 速率
MUL BINRATE ; = 乘积
ADD AX, ADJUST ; 舍入乘积
ADC DX, 00
CMP DX, SHIFT ; 对于 DIV, 乘积
JB E50 ; 太大吗?
E40: XOR AX, AX ; 若是, 清除 AX
JMP E70 ; 并退出
E50: CMP ADJUST, 00 ; 需要移位吗?
JZ E80 ; 否, 通过
DIV SHIFT ; 移位乘积
E70: XOR DX, DX ; 清除余数
E80: RET
E10MULT ENDP

;
; 把乘积转换为 ASCII:
; -----
F10PROD PROC NEAR ; 使用 AX, DX, SI
LEA SI, ASCPROD+7 ; 在 ASCII 乘积中
MOV BYTE PTR[SI], '.' ; 设置十进制小数点
ADD SI, NODECIMS ; 设置右边起点
F30: CMP BYTE PTR[SI], '.' ; 如果在十进制小数点上,
JNE F40 ; 则通过
DEC SI
F40: CMP DX, 00 ; 使用 DX:AX < 10,
JNZ F50 ; 操作完成
CMP AX, 0010
JB F60
F50: DIV TENWD ; 余数是 ASCII 数字
OR DL, 30H
MOV [SI], DL ; 存 ASCII 字符
DEC SI
SUB DX, DX ; 清余数
JMP F30
F60: OR AL, 30H ; 存最后的 ASCII
MOV [SI], AL ; 字符
RET
F10PROD ENDP

;
; 为显示而格式化乘积:
; -----
G10FORMAT PROC NEAR ; 使用 BP, CX, SI
MOV COL, LEFTCOL+3 ; 设置列
MOV CX, 09 ; ASCPROD 的长度

```

图 13-5 续


```

G20:    LEA    SI, ASCPROD      ; 清除
        CMP    BYTE PTR[SI], 30H ; 在 ASCPROD 中
        JNE    G30            ; 前导零
        MOV    BYTE PTR[SI], 20H ; 成为空白
        INC    SI
        LOOP   G20
G30:    LEA    BP, PROMPT3      ; 乘积的提示符
        MOV    CX, 20          ; 字符数
        CALL   K10DISPLY       ; 显示提示符
        RET
G10FORMAT ENDP
;
; 用户暂停, 按任意键退出:
; -----
H10PAUSE PROC NEAR            ; 使用 AX, CX, BP
        MOV    COL, 20        ; 设置光标
        MOV    ROW, 22        ; 位置
        LEA    BP, PROMPT4    ; 用户提示符
        MOV    CX, 40         ; 字符数
        CALL   K10DISPLY       ; 显示信息
        MOV    AH, 10H        ; 请求来自
        INT    16H            ; 键盘的回答
        RET
H10PAUSE ENDP
;
; 把 ASCII 量和速率转换成二进制在入口处
; SI 设置成量/速率的地址:
; -----
J10ASCBIN PROC NEAR          ; 使用 AX, BX, SI
        MOV    MULT10, 0001    ; 初始化
        MOV    BINPROD, 00
        MOV    DECIND, 00
        XOR    BX, BX          ; 清除 BX
J20:    MOV    AL, [SI]         ; 取得 ASCII 字符
        CMP    AL, '.'         ; 十进制小数点吗?
        JNE    J30            ; 否
        MOV    DECIND, 01      ; 是, 设指示符
        JMP    J40
J30:    AND    AX, 000FH        ; 强制 AH = 0
        MUL    MULT10          ; 乘以因子
        ADD    BINPROD, AX     ; 加二进制值
        MOV    AX, MULT10      ; 计算下一个
        IMUL   AX, 10          ; 因子 × 10
        MOV    MULT10, AX
        CMP    DECIND, 00      ; 到达十进制小数点了吗?
        JNZ    J40            ; 否
        INC    BX              ; 是, 加到计数中
J40:    DEC    SI              ; 下一个字符
        LOOP   J20
        CMP    DECIND, 00      ; 循环结束
        JZ     J90             ; 任何十进制小数点吗?
        ADD    NODECIMS, BX    ; 是, 加到总和中
J90:    RET
J10ASCBIN ENDP
;
; 显示字符, 设置属性:
; -----
K10DISPLY PROC NEAR          ; BP, CX 在入口设置
        MOV    AX, 1301H      ; 请求显示行
        MOV    BX, 0016H      ; 页和属性
        MOV    DH, ROW        ; 屏幕行
        MOV    DL, COL        ; 与列
        INT    10H
        RET
K10DISPLY ENDP
;
; 上卷显示窗口, 设置属性:
; -----
Q10WINDOW PROC NEAR
        MOV    AX, 0605H      ; 5行

```

图 13-5 续

```

MOV     BH,16H           ;属性
MOV     CH,TOPROW        ;左上角
MOV     CL,LEFTCOL        ;
MOV     DH,BOTROW        ;右下角
MOV     DL,RIGHTCOL       ;
INT     10H
RET
Q10WINDOW ENDP
END      A10MAIN

```

图 13-5 续

- G10FORMAT 把导前的零清除为空白，并调用 K10DISPLY 显示该值。
- H10PAUSE 提示键盘输入。(在通知程序中止处理而结束时，按<Esc>键。)
- J10ASCBIN 把 ASCII 转换成二进制(量和速率的公用例行程序)，并在输入的数据中确定十进制小数位的数量。
- K10DISPLY 在屏幕上显示数据。
- Q10WINDOW 在屏幕中部开一个窗口，用来显示量、速率和值。

1. 限制。这个程序的一个限制是：在量和速率中的十进制小数位的总数必须是 6 或小于 6。另外一个限制是乘积的大小。假定量和速率的十进制小数位的总数超过 6，或假定乘积超过约 6533.50，那么程序就要把乘积清除为 0。实际上，程序会打印警告信息或包含了克服这些限制的过程。

2. 出错检查。程序是为用户而不是为程序员设计的，它不仅会产生一些警告信息，而且应该确认量和速率的有效性。仅有的有效字符是 0 到 9 和一个十进制的小数点。对于任何其他字符，程序都将显示一个信息并重新显示输入提示符。在这一点上，XALT 是非常有用的指令，在第 14 章会涉及到它。

在练习中，用所有可能的条件严格地测试你的程序，比如零值，非常大的值和非常小的值，以及负值。

3. 负值。某些应用会包括负值，特别是对于一些反向的和修正的输入。你可以允许负的符号跟在一个值的后面，比如 12.34-，或在值的前面，如-12.34，然后程序在转换成二进制期间可以对这个负的符号进行检查。另一方面，你可能想留下二进制数的正值，再简单地设置一个指示符去记录该值是负。当该算术运算完成时，假如需要的话，该程序可以把一个负的符号插入到 ASCII 字段的左边或右边去。

为了使二进制数成为负值，照例要把 ASCII 输入转换成二进制(见在第 12 章中有关改变二进制字段符号的“符号变反”一节)，并且注意使用 IMUL 和 IDIV 去处理带符号数据。对于负数的舍入，是用减 5 代替加 5。

13.8 要 点

- ASCII 字段要求每个字符一个字节。对于一个数字字段，最右边半个字节是数字，而最左边半个字节是 3。
- 把 ASCII 数最左边的一些 3 清除为 0，则把该字段转换成非压缩的二进制(BCD)格式。

- 压缩 ASCII 字符成每个字节 2 个数位，则把该字段转换成压缩的二-十进制(BCD)数据。
- 在 ASCII 加法之后，用 AAA 来调整答案；在 ASCII 减法之后，用 AAS 来调整答案。
- 在 ASCII 乘法之前，被乘数和乘数应该用把最左边的十六进制 3 清除为 0 的办法转换成非压缩的 BCD。乘法之后，用 AAM 来调整乘积。
- 在 ASCII 除法之前，被除数与除数应该用清除最左边十六进制 3 的办法转换成非压缩的 BCD，而 AAD 则用于调整被除数。
- 对于大多数算术运算来说，ASCII 数应该转换成二进制。用于这一目的的有效 ASCII 字符是 30H 到 39H，十进制的小数点，以及有可能的负的符号。

13.9 习 题

13-1. 假设 BX 的内容是 ASCII 9(0039H)，而 DX 的内容是 ASCII 5(0035H)。说明以下不相关操作的结果：

(a) ADD BX, 34H (b) ADD BX, DX (c) SUB BX, DX (d) SUB BX, 0CH
AAA AAA AAS AAS

13-2. 使用十六进制表示法把十进制值 4127 表示成以下格式：

(a) ASCII, (b) 非压缩的 BCD, (c) 压缩的 BCD。

13-3. 一个名为 BCDVAL 的非压缩 BCD 字段的内容是 01060803H。编写一个循环，使该字段的内容成为正确的 ASCII 31363833H。

13-4. 名为 ASCVAL1 的字段的内容是 ASCII 十进制值 174，而另一个名为 ASCVAL2 字段的内容是 ASCII 4。编写指令进行 ASCII 数相乘，并把乘积存放在 ASCPROD 中。

13-5. 使用如 13-4 题同样的字段，把 ASCVAL1 除以 ASCVAL2 并把商存放在 ASCQUOT 中。

13-6. 对以下操作提供手工计算：(a)把 ASCII 十进制值 29765 转换成二进制值，并以十六进制格式表示其结果；(b)反过来再把十六进制值转换成 ASCII 值。

13-7. 编写并测试一个程序：(a)在 EAX, EBX, ECX 和 EDX 中插入二进制值，(b)以 ASCII 格式显示每个寄存器的值。

13-8. 编写并测试一个程序：(a)从键盘接受 2 个数字值，(b)把 2 个 ASCII 值转换成二进制格式，(c)二进制值相加，(d)把二进制和转换成 ASCII 格式，(e)显示 ASCII 和。要考虑到任何成对的数的输入。为指明不再输入数据，用户只需按 <Enter> 键。

目的：说明对定义表格的要求，实现对表格的查找，以及表格项目的排序。

14.1 引言

许多程序都需要应用一些**表格或数组**，它们包含这样一些数据，如名字、货名、量和速率等。本章从定义某些常用的表格开始，然后再涉及全程查找它们的方法。有关查找表格的技术易受定义表格所用方法的制约，并且许多定义与查找表格的方法可能并不是这里所给出那些方法。表格的定义和使用，基本上和已经学过的应用有关。其他通用特性是排序的使用。排序是重新排列表格中的数据顺序，以及地址表与展开表的使用。

在本章中，介绍的唯一指令是 XLAT(换码)。

14.2 定义表格

为了便于对表格进行全程查找，大多数表格都按一致的样式，它的每个项目是按同样的格式(字或数字)，同样的长度定义，并且按升序或降序来排列。

在本书中，常用的表格是堆栈的定义，它是一个 64 个未初始化字的表格，其中名字 STACK 指向该表的第一个字，如 STACK DW 64 DUP(?)。

下面两个表格 MONTH_TBL 和 CUST_TBL 分别初始化字符和数字值。MONTH_TBL 定义月份的字母缩写，而 CUST_TBL 则是定义一个用户编号表：

```
MONTH_TBL DB 'Jan', 'Feb', 'Mar', ..., 'Dec'
CUST_TBL  DB 205, 208, 209, 212, 215, 224, ...
```

MONTH_TBL 表中的所有项目都是 3 个字符。但是，尽管 CUST_TBL 的所有项目是按 3 个数字定义的，汇编程序还是把十进制数转换成二进制格式，而且其值不能超过 255，因为它们是按每个数一个字节存放的。

表格还可以包含数字与字符的混合形式，这只要与它们的定义相符合就行。在以下的库存品项目的表格中，每个数字项目(库存品号)是 2 个数位(一个字节)，而每个字符项目(库存品名称)是 9 个字节：

```
STOCK_TBL DB 12, 'Computer', 14 'Paper...', 17 'Diskettes', ...
```

紧跟着名称“Paper”后的4个点表示的是将有空格存在；也就是说，它们不是点，而是紧跟名称的空格。为清楚起见，可以把每个表格项目对，写在单独的一行上：

```
STOCK_TBL DB 12, 'Computer'
           DB 14, 'Paper...'
           DB 17, 'Diskettes'
           ...
```

下一个例子定义有100个项目的表格，每个项目被初始化为15个空白(总共是1500个字节)：

```
STOCK_TBL DB 100 DUP(15 DUP(' '))
```

程序可以使用这个表格去存放多达100个值，这些值是内部产生的，或者使用这个表格去存放多达100个项目的内容，它们是从键盘接受的或是从磁盘文件读出的。

在实际情况下，许多程序是表格驱动的。也就是说，表格是作为磁盘文件存放的，任意多的程序可以要求去处理它。为此目的，程序可以从磁盘把一个表格文件读入到按用途定义的一个“空”表格中。有关这一做法的原因在于，表格内容随时都可能改变。如果每个程序都定义它自己的表格，那么任何改变都要求重新定义表格并重新汇编该程序。由于表格文件在磁盘上，所以对于表格的改变可以简单地只牵涉到改变文件的内容。第17章给出一个表格文件的例子。

类型(TYPE)、长度(LENGTH)与规模(SIZE)操作符

汇编程序提供许多专门的操作符，你会发现它们是很有用的。例如，表格长度随时可能改变，而可能不得不修改程序去说明新的定义，并要加上例行程序去检查表格的结束。使用TYPE、LENGTH和SIZE操作符，可能有助于减少那些必须改变的指令数。

考虑这个有12字表格的定义：

```
RAIN_TBL DW 12 DUP(?) ; 有12个字的表格
```

程序可以使用TYPE操作符去确定定义(在这种情况下是DW)，LENGTH操作符去确定DUP因子(12)，而用SIZE操作符确定字节数($12 \times 2 = 24$)。以下例子说明这3个操作符：

```
MOV AX, TYPE RAIN_TBL ; AX=0002H(2个字节)
MOV BX, LENGTH RAIN_TBL ; BX=000CH(12个字节)
MOV CX, SIZE RAIN_TBL ; CX=0018H(24个字节)
```

可以用LENGTH和SIZE的值返回去结束一个表格的查找或排序。例如，如果SI寄存器包含查找的增量偏移地址，那就可以使用

```
CMP SI, SIZE RAIN_TBL
```

测试这个偏移值。

第25章会详细讨论TYPE、LENGTH和SIZE操作符，现在让我们来研究一下在程序中使用表格的不同方法。

14.3 表格项目的直接寻址

假设用户输入一个数字月份(比如 03), 而程序把它转换成字母格式——在这种情况下就是 March(3 月)。例行程序实现这一转换要包括定义一个字母月份表, 全部项是等长的。每个项目的长度应该是最长的名字 September(9 月)的长度, 格式是:

```
MONTH_TBL DB 'January..'
           DB 'February..'
           DB 'March....'
           ...
           DB 'December..'
```

项目 'January' 在 MONTH_TBL+00 处, 'February' 在 MONTH_TBL+09 处, 'March' 在 MONTH_TBL+18 处, 以此类推。假设用户键入 3(对于 March), 程序要在表格中对它定位。该程序必须按以下步骤执行:

1. 把月份项目从 ASCII 33 转换成二进制 3。
2. 从这个数中减去 1: $3-1=2$ (由于月份 01 在 MONTH_TBL+00 处)。
3. 这个新的数乘以 9(每个项目的长度): $2 \times 9=18$ 。
4. 把这个乘积(18)加到 MONTH_TBL 的地址上, 该结果就是所要求项目的地址:

MONTH_TBL+18, "March" 从这里开始。

这一技术称为直接表格寻址。由于算法直接计算所要求的表格地址, 所以不必要在表格中定义数字的月份, 并且程序也不必要连续地全程查找该表格。

14.3.1 直接寻址, 例 1: 月份表

在图 14-1 中的部分程序提供一个按月份名直接访问表格的例子。该程序假定 12(December) 作为输入, 并把月份从 ASCII 转换成二进制格式(按照已在第 13 章讨论过的转换方法):

```
LEN_ENTRY EQU 9           ; 表格项目长度
MONTH_IN DB '12'          ; ASCII '3132'
MONTH_TBL DB 'January ', 'February ', 'March '
           DB 'April ', 'May ', 'June '
           DB 'July ', 'August ', 'September '
           DB 'October ', 'November ', 'December '
;
; 把 ASCII 月份转换成二进制:
XOR WORD PTR MONTH_IN, 3030H ; 清除 ASCII 3
MOVZX AX, MONTH_IN           ; 左边的数位
IMUL AX, 10                  ; 乘以 10 并和
ADD AL, MONTH_IN+1           ; 右边的数位相加
;
; 定位表中月份:
DEC AL                       ; 校正表格
IMUL AX, LEN_ENTRY           ; AL 乘以 9
LEA BP, MONTH_TBL            ; 把偏移值加到
ADD BP, AX                    ; 表格的地址上
;
; 显示字母月份:
;
MOV AX, 1301H                ; 已装入 ES:BP
MOV BX, 0016H                ; 请求显示
MOV CX, LEN_ENTRY            ; 页; 属性
MOV DX, 0812H                ; 9 个字节
INT 10H                      ; 行; 列
...
```

图 14-1 直接表格寻址: 例 1

```

初始输入月份(12) -      3132H
用 3030H XOR =          0102H
月份左边字节乘以 10 =    0AH
加上月份右边的字节 =    0CH(十进制的 12)

```

该程序确定在表格中的月份的实际位置:

```

从在 AX 中的月份里减 1 =    000BH(十进制 11)
乘以 9(项目的长度) -      0063H(十进制 99)
加上表格的地址 =          MONTH_TBL+63H

```

改进这一程序的一种方法是从键盘接受数字月份,并核实那个值是包括在 01 和 12 之间的。

14.3.2 直接寻址,例 2: 月份的日期表

在图 14-2 中的部分程序从系统中取回今天的日期并加以显示。INT 21H 功能 2AH 提供以下二进制值:

```

AL=星期(其中星期日=0)      DH=月(01-12)
CX=年(本程序未用)          DL=日(01-31)

```

该程序用这些返回值以“Wednesday”,“September”,以及“12”的形式去显示字母表示的星期几和月份。为此,程序定义一个名为 DAYS_TBL 的从 Sunday 开始的星期的表格,以及名为 MONTH_TBL 的从 January 开始的月份的表格。

在 DAYS_TBL 和 MONTH_TBL 中的项目是 9 个字节长,每个说明的右边用空白填入。该程序把星期乘以 9(在 DAYS_TBL 中每个项目的长度)。乘积是表中的一个偏移值,例如 Sunday 是在 DAYS_TBL+0 处,Monday 是在 DAYS_TBL+9 处,以此类推。日期是直接从表格显示的。

该程序使月份减 1,例如,使月份 01 变成 MONTH_TBL 中的项目 0。然后它再把月份乘以 9(在 MONTH_TBL 中每个项目的长度),程序直接从表格显示月份。

该程序将月中的日除以 10,把它从二进制转换成 ASCII 格式。由于日的最大值是 31,所以商与余数可能各自仅为一个数位(例如,31 被 10 除得到一个 3 的商和一个余数 1)。程序显示 2 个字符中的每一个,包括小于 10 的日期的前导 0(删除前导 0,会使程序有少量的改变)。

尽管直接表格寻址是非常有效的,但它工作得最好的情况是当项目是按照顺序的,并且是按可预测的次序排列的时候。因此对于按这样的次序:01,02,03,...,或 106,107,108,...,以至于 05,10,15,...,它是能良好地工作的。但是,只有少数应用才能提供这么整齐排列的表格值。下一节要研究的表格的值是按顺序的,但并不是按可预测的次序。

```

LEN_ENTRY EQU 9 ; 表格项目长度
DAYOFMON DW 00 ;
SAVEDAY DB ? ;
SAVEMON DB ? ;
TEN DB 10
ROW DB 10 ; 屏幕行
COLUMN DB 30 ; 与列
DAYS_TBL DB 'Sunday', 'Monday'
DB 'Tuesday', 'Wednesday'
DB 'Thursday', 'Friday'
DB 'Saturday'
MONTH_TBL DB 'January', 'February', 'March'
DB 'April', 'May', 'June'
DB 'July', 'August', 'September'
DB 'October', 'November', 'December'
.386 ;
MOV AH, 2AH ; 取得今天的日期
INT 21H
MOV SAVEMON, DH ; 保存月份
MOV SAVEDAY, DL ; 保存月日
; 显示星期:
MOV AH, 0
IMUL AX, LEN_ENTRY ; 日×项目长度
LEA BP, DAYS_TBL ; 表格地址
ADD BP, AX ; 加偏移值
MOV CX, LEN_ENTRY ; 长度
CALL B10DISPLY ;
; 显示月份:
MOVZX AX, SAVEMON ;
DEC AX ; 月份减 1
IMUL AX, LEN_ENTRY ; 月份×项目长度
LEA BP, MONTH_TBL ; 表格地址
ADD BP, AX ; 加偏移值
MOV CX, LEN_ENTRY ; 长度
CALL B10DISPLY ;
; 显示日:
MOVZX AX, SAVEDAY ;
DIV TEN ; 把日从二进制转换成
OR AX, 3030H ; ASCII
MOV DAYOFMON, AX ; 保存ASCII日
LEA BP, DAYOFMON ;
MOV CX, 2 ; 字符数
CALL B10DISPLY ;
...
; 公用显示例行程序:
B10DISPLY PROC NEAR ; BP, CX 在入口设置
PUSHA ; 保存寄存器
MOV AX, 1301H ; 请求显示
MOV BX, 0016H ; 页: 属性
MOV DH, ROW ; 屏幕行
MOV DL, COLUMN ; 与列
INT 10H
INC ROW ; 下一行
POPA ; 恢复寄存器
RET
B10DISPLY ENDP

```

图 14-2 直接表格寻址: 例 2

14.4 查找表格

某些表格是由没有明显模式的独特的编号组成的。典型的例子是库存品项的表格，它的编号是不按顺序的，比如 034, 038, 041, 139, 以及 145。另一类表格，例如收入税表，包含值的范围。以下几节研究这些类型的表格，以及对查找它们的要求。

14.4.1 具有独特项目的表格

大多数商业库存品项的编号通常都是不按顺序的。更确切地说，它们往往根据类型分组，这些类型多半有前导数，用以指明设备或用具，或者指明它是存放在某个部门的。还有，某些项会随时从库存品中删除并加进另一些项。作为一个例子，让我们定义一个表格，它具有库存品号及其相关的说明。这些可以按单独的表格加以定义，比如

```
STOCK_NO    DB  '05', '10', '12', ...
STOCK_DESC  DB  'Excavators', 'Lifters...', 'Presses...', ...
```

查找中的每一步都使第一个表格的地址加 2(在 STOCK_NO 中每个项目的长度)，而第二个表格的地址加 10(在 STOCK_DESC 中每个项目的长度)。或者，过程可以保持所执行的循环次数的计数值，在找到与某一个关键库存号相匹配时，把该计数值乘以 10，再利用这个乘积作为 STOCK_DESC 的地址偏移值。

另一方面，把库存品号和说明定义在同一表格中可能比较清楚，每对项目有一行：

```
STOCK_TBL   DB  '05', 'Excavators'
              DB  '10', 'Lifters...'
              DB  '12', 'Presses...'
              ...
```

图 14-3 的部分程序用 6 对库存品号与说明定义这个表格。查找例行程序开始把输入的库存品号 STOCK_IN 的第一个字节和在表格中库存品号的第一个字节进行比较。比较结果可能是低、高或相等。

1. 低于。如果第一或第二个字节的比较结果是低于，则程序确定该库存品号不在表格内，并在 A40 处显示一个出错信息(未编码)。例如，程序把输入库存品项 01 和表格项 05 作比较：第一个字节是相等的，但由于第二个字节是低于，所以程序确定该项不在表格中。

2. 高于。如果第一个字节或第二个字节的比较结果是高于，则程序继续查找。为了把输入库存品项和在表格中的下一个库存品项进行比较，程序要使 SI 中包含的表格地址增量。例如，程序比较输入库存品项 06 和表格项 05，第一个字节是相等的，但第二个字节是高于，所以它把输入的项和表格中的下一个项相比较：库存品项 06 和表格项 10。第一个字节是低于，所以程序确定该项目不在表格中。

3. 等于。如果第一个字节和第二个字节是相等的，那么可以找到库存品号。例如，程序比较输入库存品项 10 和表格项 05。第一个字节是高于，所以它比较输入库存品项和表格中的下一项：库存品项 10 和表格项 10。由于第一个字节是相等的，并且第二个字节也是相等的，所以程序找到了该项目，在 A50 处直接显示来自表格的说明。

循环查找 6 次比较的最大值。如果循环次数超过 6，那么便能知道库存品号不在表格中。

表格还可以定义单价。用户键入库存品号和销量。程序可以将表格中的库存品项定位，计算销售总额(销量乘以单价)，并显示说明和销售总额。

在图 14-3 中，库存品号是 2 个字符，而说明是 10 个字符。程序设计的细节由于项目数与项目长度的不同会有所变化。例如，为比较 3 字节的字段，可以使用 REPE CMPSB，尽管 REPE 包括 LOOP 时已使用过的 CX 寄存器。

LEN_STKNO	EQU	02	; 库存品号
LEN_DESCR	EQU	10	; 与说明的长度
STOCKN_IN	DB	'12'	; 输入的库存品号
STOCK_TBL	DB	'05','Excavators'	; 表格起点
	DB	'10','Lifters'	
	DB	'12','Presses'	
	DB	'15','Valves'	
	DB	'23','Processors'	
	DB	'27','Pumps'	; 表格终点

	...		
	MOV	CX,06	; 初始化
	LEA	SI,STOCK_TBL	; 比较
A20:			
	MOV	AL,STOCKN_IN	
	CMP	AL,[SI]	; 库存品 # (1): 表格
	JNE	A30	; 不相等, 退出
	MOV	AL,STOCKN_IN+1	; 相等,
	CMP	AL,[SI+1]	; 库存品 # (2): 表格 +1
	JE	A50	; 相等, 找到
A30:	JB	A40	; 低于, 不在表格中
	ADD	SI,LEN_STKNO	; 高于, 得到下一个
	ADD	SI,LEN_DESCR	; 项目
	LOOP	A20	
A40:			; 不在表格中,
	...		; 显示出错信息
	JMP	A90	; 并退出
A50:			
	INC	SI	
	INC	SI	
	MOV	AX,1301H	; 提取说明
	MOV	BF,SI	; 请求显示
	MOV	BX,0061H	; 库存品说明
	MOV	CX,LEN_DESCR	; 页+属性
	MOV	DX,0812H	; 10 个字符
	INT	10H	; 行: 列
A90:	...		

图 14-3 使用 CMP 查找表格

14.4.2 带范围的表格

所得税提供一个带范围值的表格的典型例子。考虑以下应纳税收入，税率和调节系数的假想表格：

应纳税收入(\$)	税率	调节系数
0-1,000.00	.10	000.00
1,000.01-2,500.00	.15	050.00
2,500.01-4,250.00	.18	125.00
4,250.01-6,000.00	.20	260.00
6,000.01 及更高	.23	390.00

在这个税表中，税率随应纳税收入的增加而增加。调节系数对于我们计算的高税率的税进行补偿，而低税率则适用于较低收入水平。应纳税收入的项目包括每一等级的最高收入：

TVATBL	DD	100000, 10, 00000
	DD	250000, 15, 05000
	DD	425000, 18, 12500
	DD	600000, 20, 26000
	DD	999999, 23, 39000

为实现该表格的查找,程序要把纳税人的实际应纳税收入和从表格第一项开始的值进行比较,并根据比较的结果做以下事情:

- 高于:还没有找到,为表格中的下一项增量。
- 低于或等于:找到了,使用相应的税率和调节系数。按(应纳税收入×表中的税率),即调节系数来计算扣除的税额。注意,表格中最后一项是最大值(999999),在这里总能正确地强制结束查找。

14.4.3 使用串比较查找表格

REPE CMPS 对于项目编号有 2 个或更多字符长的比较是有用的。图 14-4 的部分程序定义了 STOCK_TBL,但这次库存品号被修改成了 3 字节。在表格中的最后一项是库存品项 '999' (最大可能的库存品号),它强制结束查找。该程序使用了循环强制结束查找,但 REPE 使 CX 不能用于 LOOP。查找例行程序把 STOCKN_IN(任意定义为 123)和每个表格项目进行比较,如下所示:

STOCKN_IN	表的项目	比较结果
123	035	高于,检查下一项
123	038	高于,检查下一项
123	049	高于,检查下一项
123	102	高于,检查下一项
123	123	等于,项目被找到

	LEN_STKNO	EQU	03	; 库存品号长度
	LEN_DESCR	EQU	10	; 和说明的长度
0000	STOCKN_IN	DB	'123'	
0003	STOCK_TBL	DB	'035','Excavators'	; 表格
0010		DB	'038','Lifters'	
001D		DB	'049','Presses'	
002A		DB	'102','Valves'	
0037		DB	'123','Processors'	
0044		DB	'127','Pumps'	
0051		DB	'999', 10 DUP(' ')	

;				
	...			
	CLD			
	LEA	DI,STOCK_TBL	; 初始化表格地址	
A20:	MOV	CX,LEN_STKNO	; 设置成比较 3 个字节	
	LEA	SI,STOCKN_IN	; 初始化库存品号地址	
	REPE	CMPSB	; 库存品号: 表格	
	JE	A30	; 相等, 退出	
	JB	A40	; 低于, 不在表格中	
	ADD	DI,CX	; 把 CX 值加到偏移值上	
	ADD	DI,LEN_DESCR	; 下一个表格项目	
	JMP	A20		
A30:	MOV	AX,1301H	; 请求显示	
	MOV	BP,DI	; 库存品说明	
	MOV	BX,0061H	; 页: 属性	
	MOV	CX,LEN_DESCR	; 10 个字符	
	MOV	DX,0812H	; 行: 列	
	INT	10H		
	JMP	A90		
A40:	...			
;				
A90:	...	< 显示出错信息 >		

图 14-4 使用 CMPSB 查找表格

程序首先初始化 DI 为 STOCK_TBL 的偏移地址(003),CX 初始化为每个库存品项的长

度(03), SI 初始化为 STOCKN_IN 的偏移地址(000)。只要被比较的字节包含相等的值, CMPSB 操作就进行字节与字节的比较, 并且 DI 与 SI 为下一对字节而自动增量。第一个表格项目的比较(123: 035)是在第一个字节后用高于的比较结束的, DI 是 004, SI 是 001, 而 CX 是 02。

对于第二个表格项目的比较, DI 应当是 010, 而 SI 应当是 000, 对 SI 的校正是简单地重新装入 STOCKN_IN 的地址。然而, 为了校正在 DI 中的表格项目的地址, 增量取决于比较是结束于 1 个、2 个还是 3 个字节之后。CX 包含剩下的未比较的字节数, 在这种情况下是 02。加 CX 的值, 再加上库存品说明的长度(以前被比较的库存品项目的说明)给出下一个表格项目的偏移值, 如下所示:

CMPSB 之后 DI 中的地址:	004H
加上 CX 中余下的长度:	+ 02H
加上库存品说明的长度:	+ 0AH
表格中的下一个偏移地址:	010H

由于 CX 包含剩下的未比较的字节数(如果有的话), 所以对于所有情况来说, 运算还在进行, 直到 1 次、2 次或 3 次比较之后结束。在相等比较时, CX 是 00, 而 DI 已经增加到了所要求说明的地址。程序直接显示来自表格的说明。

14.4.4 具有可变长度项目的表格

定义一个具有可变长度项目的表格是可能的。专门的定界符字符(如 00H)可以跟在每个表格项目的后面, 而 FFH 可以标记表格的结束。SCAS 指令适用于定界符的扫描。但必须保证在项目内没有包含定界符的位配置, 例如, 一个算术的二进制总数计算可能包含任何可能的位配置, 包括 00H 和 FFH。

14.5 XLAT(换码)指令

XLAT 指令把一个字节的位配置转换成另一个预先定义的配置。例如, 使用 XLAT 验证数据项的内容或者加密数据。XLAT 的格式是

[label:]	XLAT	; 没有操作数
-----------	------	---------

为了使用 XLAT, 需要定义一个换码表, 它由全部 256 种可能的字符构成。XLAT 要求表格的地址在 BX 中, 而被转换的字节是在 AL 中。

下面的例子是把 ASCII 数 0-9 转换成 EBCDIC 格式, 它适用于 IBM 大型计算机。由于 ASCII 格式表示的 0-9 是 30-39, 而用 EBCDIC 表示是 F0-F9, 可以使用 OR 操作进行这种转换。但是, 可以任意地把 ASCII 负的符号(2D)和十进制小数点(2E)转换为 EBCDIC(分别是 60 和 4B), 并且所有其他字符都是空格(在 EBCDIC 格式里它是 40H)。在换码表中, EBCDIC 码是按 ASCII 位置定义的, 即 EBCDIC 字符是在 ASCII 单元里, 而 EBCDIC 负的符号、十进制小数点和空格是在另一些单元里。由于数 0 是 ASCII 30H, 所以 EBCDIC 数是从表格的 30H 或十进制 48 单元开始的:

```

XLAT_TBL DB 45 DUP(40H)           ; 换码表
          DB 60, 4BH
          DB 40H
          DB 0F0H, 0F1H, 0F2H, 0F3H, 0F4H
          DB 0F5H, 0F6H, 0F7H, 0F8H, 0F9H
          DB 198 DUP(40H)

```

注意：在 XLAT_TBL 中的第一个 DB 定义 45 个字节，它的地址是 XLAT_TBL+00 到 XLAT_TBL+44。第二个 DB 定义的数据从 XLAT_TBL+45 开始，以此类推。

XLAT 使用 AL 的值作为偏移地址，实际上，BX 包含的是表格的起始地址，而 AL 包含的是表格内的偏移值。例如，如果 AL 值是 00，那么表格地址将是 XLAT_TBL+0(XLAT_TBL 的第一个字节是 40H)。XLAT 会用来自表格的 40H 取代 AL 中的 00。如果 AL 值是 32H(十进制 50)，那么表格地址是 XLAT_TBL+50，这一单元包含 F2(EBCDIC 的 2)，会被 XLAT 插入到 AL 中。

下面的程序片段是经过 6 字节 ASCII 字段 ASC_NO 的循环，用 XLAT_TBL 把这个字段转换成 EBCDIC 格式。最初，ASC_NO 的内容是 -31.5 跟着一个空格或是十六进制的 2D33312E3520。在循环结束时，EBC_NO 的内容是十六进制的 60F3F14BF540，可以借助于 DEBUG 加以验证。

```

ASC_NO DB ' -31.5 '           ; 要转换的 ASCII 项
EBC_NO DB 6 DUP(' ')         ; 转换了的 EBCDIC 项
...
LEA SI, ASC_NO                ; ASCII 数的地址
LEA DI, EBC_NO                ; EBCDIC 数的地址
MOV CX, 06                    ; 项的长度
LEA BX, XLAT_TBL              ; 表格的地址

L10:
LODSB                         ; 取 ASCII 字符放在 AL 中
XLAT                           ; 转换字符
STOSB                         ; AL 存入 EBC_NO
LOOP L10                      ; 重复 6 次

```

程序：显示十六进制与 ASCII 字符

在图 14-5 中的部分程序显示全部 256 个十六进制值(00-FF)，包括大多数与它们相关的 ASCII 符号，例如 ASCII 符号 S 及其十六进制表示：53。完整的显示是以 16 乘 16 矩阵形式出现在屏幕上的：

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
.
.
.
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

如图 8-1 所示，显示 ASCII 字符不会产生严重问题。但是，显示 ASCII 值的十六进制表

示就要复杂一些。例如，必须把 00H 转换成 3030H，01H 转换成 3031H，以此类推。这就是说，01H 要作为 2 个字符 01 来显示。

```

ROW      DB      02
DISPROW  DB      16 DUP(5 DUP(' '))
HEXCTR   DB      00
XLATAB   DB      30H,31H,32H,33H,34H,35H,36H,37H,38H,39H
          DB      41H,42H,43H,44H,45H,46H

.386 ;
          CALL    clear_screen      ;标准过程
          LEA     SI,DISPROW         ;初始化显示行
A20:      CALL    B10HEX             ;转换与
          CALL    C10DISPLY          ;显示
          CMP     HEXCTR,0FFH        ;最后的十六进制值 (FF) 吗？
          JE      A90                ;是，退出
          INC     HEXCTR             ;否，下一个十六进制值增量
          JMP     A20
          ...
;
B10HEX    PROC    把 ASCII 转换成十六进制：
          NEAR
          MOVZX   AX,HEXCTR          ;取 AX 中的十六进制对
          SHR     AX,04              ;移掉右边的十六进制数位
          LEA     BX,XLATAB          ;设置表格地址
          XLAT
          MOV     [SI],AL            ;转换十六进制数位
          MOV     AL,HEXCTR          ;把左边字符存
          AND     AL,0FH             ;在显示行上
          XLAT
          MOV     {SI}+1,AL          ;消除左边十六进制数位
          RET
B10HEX    ENDP
;
C10DISPLY PROC    显示十六进制字符：
          NEAR
          MOV     AL,HEXCTR          ;取字符
          MOV     [SI]+3,AL
          CMP     AL,07H             ;比 7 低吗？
          JB      C20                ;是，往下进行
          CMP     AL,10H             ;高于/等于 16 吗？
          JAE     C20                ;是，往下进行
          MOV     BYTE PTR [SI]+3,20H ;否则，强制为空白
C20:      ADD     SI,05              ;行中的下一单元
          LEA     DI,DISPROW+80
          CMP     DI,SI
          JNE     C90                ;填满行了吗？
          ;否，退出
          MOV     AX,1300H           ;请求显示
          MOV     BX,0031H           ;页和属性
          LEA     BP,DISPROW         ;数据
          MOV     CX,80              ;行的长度
          MOV     DH,ROW             ;行
          MOV     DL,00              ;列
          INT     10H
          INC     ROW                ;下一行
          LEA     SI,DISPROW         ;重新初始化
C90:      RET
C10DISPLY ENDP

```

图 14.5 显示 ASCII 和十六进制值

程序最初用 00H 定义 HEXCTR，并使 256 个 ASCII 字符的每一个依次加 1。过程 B10HEX 把 HEXCTR 分割为它的 2 个十六进制数位。例如，HEXCTR 的内容是 4FH，例行程序将抽取一个十六进制的 4，由 XLAT 用于转换。返回到 AL 的值是 34H。然后例行程序再抽取 F 并把它转换为 46H。结果 3446H 在屏幕上显示成 4F。

过程 C10DISPLY 把非 ASCII 字符转换成空格。由于 INT 10H 功能 13H 能对退格 (Backspace) 和其他控制字符起作用，所以程序把它们改成空格。过程显示一个 16 字符的整行，并在显示第 16 行之后结束。

有许多方法把十六进制数字转换为 ASCII 字符，例如，可以用移位与比较。

14.6 表格项目排序

有的应用场合需要把表格中的数据按照升序或降序的顺序进行排序。例如，用户可能想要一个按升序序列的库存品说明表，或另一个按降序序列的销售代理商总销售额表。有许多表格排序例行程序，它们各不相同，有的处理速度相对较慢，但程序清晰；有的处理速度快，但难于理解。在这一节中所提供的例行程序效率是相当高的，并且适用于大多数表格的排序。

对表格进行排序的一般方法是把表格的项目与紧跟其后的项目进行比较。如果比较结果是高于，则交换该项目。继续利用这种方法，项目 1 与项目 2 相比较，项目 2 与项目 3 相比较，以此类推，直到表格的末端，其中进行交换是必须做的。要是进行了任意次交换，还要从表格的起点重复完整的处理，再一次进行项目 1 与项目 2 的比较，以此类推。在任何时候，假如进行了整个表格的处理而没有做过一次交换，那么就on知道表格已经完成了排序。

在以下的伪代码中，SWAP 是一个项，它指明是进行了交换(YES)还是没有进行交换(NO)。

```

L10:  初始化表格中最后一个项目的地址
      L20:  将 SWAP 设置成 NO
          初始化表格起点的地址
      L30:  表格项目 > 下一个项目吗?
          是: 交换项目
              设置 SWAP 为 YES
          为表格中的下一个项目增量
          是在表格的末尾吗?
          否: 转移到 L30
          是: SWAP=YES 吗?
              是: 转移到 L20 (重复排序)
              否: 结束排序

```

图 14-6 中的程序允许用户键入来自键盘的多达 30 个的人名，程序把它们存入名为 NAMETABLE 的表格中。它包括了以下过程：

- A10MAIN 调用 B10ENTER 接收来自键盘的人名，调用 C10STORE 把人名存入表格中，并且当键入所有人名时，调用 D10SORT 和 F10NAMES。
- B10ENTER 提示用户键入一个人名，接收它，并用空格填充它的右边。当键入所有人名时，用户只要按回车键，不带人名。
- C10STORE 将每个人名逐次地存入表格中。
- D10SORT 和 E10XCHNG 将人名的表格按升序序列排序。
- F10NAMES 显示已排序了的表格。

注意，表格项目全部是 20 个字节的固定长度，对可变长度数据进行排序的例行程序会更为复杂。

```

TITLE      A14NMSRT (EXE) 对从键盘输入的人名进行排序
.MODEL     SMALL
.STACK     64
.DATA
LEN_NAME   EQU      20                      ; 人名的长度
ENDADDR    DW        ?
MESSG1     DB        'Name? '
NAMECTR     DB        00
NAMESAVE   DB        20 DUP(?)
NAME_TBL   DB        30 DUP(20 DUP(' ')) ; 人名表格
ROW         DB        00                      ; 屏幕行
SWAPPED    DB        00
NAMEPAR     LABEL    BYTE                      ; 人名参数表
MAXNLEN     DB        21                      ; 最大长度
NAMELEN     DB        ?                      ; 输入字符数
NAMEFLD     DB        21 DUP(' ')           ; 输入人名
.386
;-----
.CODE
A10MAIN    PROC      FAR
MOV        AX,@data                      ; 初始化DS和
MOV        DS,AX                          ; ES 寄存器
MOV        ES,AX
CLD
CALL       Q10CLEAR                       ; 清除屏幕
LEA        DI,NAME_TBL                   ; 初始化表格地址
A20:
CALL       B10ENTER                       ; 从键盘接受人名
CMP        NAMELEN,00                    ; 还有人名吗?
JE         A30                            ; 否, 进行排序
CMP        NAMECTR,30                    ; 输入30个人名了吗?
JE         A30                            ; 是, 进行排序
CALL       C10STORE                       ; 把输入的人名存入表格中
JMP        A20                            ; 重复
A30:
CALL       Q10CLEAR                       ; 输入结束
CMP        NAMECTR,01                    ; 清除屏幕
JBE        A90                            ; 一个或没有人名输入?
JBE        A90                            ; 是, 退出
CALL       D10SORT                        ; 存入的人名排序
CALL       F10NAMES                       ; 显示已排序的人名
A90:
MOV        AX,4C00H                      ; 结束处理
INT        21H
A10MAIN    ENDP
;
; 按输入接受人名, 清除其右边:
;-----
B10ENTER    PROC      NEAR                      ; DI 按项目设置
LEA        BP,MESSG1                      ; 使用 AH, BP, BX, CX, DX
MOV        CX,06                          ; 提示符
CALL       G10DISPLY                      ; 字符数
INC        ROW                            ; 调用显示例行程序
MOV        AH,0AH                          ; 下一行
LEA        DX,NAMEPAR                      ; 接受人名
INT        21H
MOVZX      BX,NAMELEN                      ; 取字符数
MOVZX      CX,MAXNLEN                      ; 最大长度-实际长度
SUB        CX,BX                          ; 剩下的长度
B20:
MOV        NAMEFLD[BX],20H                 ; 把人名剩余部分
INC        BX                             ; 清除为空白
LOOP       B20
RET
B10ENTER    ENDP
;
; 把输入的人名存入表格:
;-----
C10STORE    PROC      NEAR                      ; 在入口设置表格的 DI
INC        NAMECTR                       ; 使用 AH, BP, BX, CX
CLD                                       ; 加到人名计数
LEA        SI,NAMEFLD                     ; 传送
MOV        CX,LEN_NAME                     ; 人名(SI)
REP        MOVSB                          ; 到表格(DI)
RET
C10STORE    ENDP

```

图 14-6 人名表格的排序


```

;
; ----- 按升序列对表格中的人名进行排序: -----
D10SORT PROC NEAR ; 在入口设置表格的 DI
SUB DI, 40 ; 使用 AX, BX, CX, DI, SI
MOV ENDADDR, DI ; 设置表格中最后人名
D20: MOV SWAPPED, 00 ; 的停止地址
LEA SI, NAME_TBL ; 设置表格
; 起点
D30: MOV CX, LEN_NAME ; 比较的长度
MOV DI, SI
ADD DI, LEN_NAME ; 比较的下一个人名
MOV AX, DI ; 保留
MOV BX, SI ; 地址
REPE CMPSB ; 与下一个人名比较
JBE D40 ; 没有交换
CALL E10XCHNG ; 交换
D40: MOV SI, AX
CMP SI, ENDADDR ; 表格末尾吗?
JBE D30 ; 否, 继续
CMP SWAPPED, 00 ; 有任何交换吗?
JNZ D20 ; 有, 继续
RET ; 无, 排序结束
D10SORT ENDP
;
; ----- 交换表格的项目: -----
E10XCHNG PROC NEAR ; 在入口设置 BX
MOV CX, LEN_NAME ; 使用 CX, DI, SI
LEA DI, NAMESAVE ; 字符数
MOV SI, BX ; 暂时地
REP MOVSB ; 保留
; 较低项
MOV CX, LEN_NAME ; 把较高项传
MOV DI, BX ; 送到
REP MOVSB ; 较低项
MOV CX, LEN_NAME ; 把保留项
LEA SI, NAMESAVE ; 传送到
REP MOVSB ; 较高项
MOV SWAPPED, 01 ; 产生信号交换
RET
E10XCHNG ENDP
;
; ----- 显示表格中已排序的人名: -----
F10NAMES PROC NEAR ; 使用 BP, CX
MOV ROW, 00 ; 初始化行
LEA BP, NAME_TBL ; 初始化表格起点
F20: MOV CX, LEN_NAME ; 字符数
CALL G10DISPLY ; 调用显示例行程序
INC ROW ; 下一行
ADD BP, LEN_NAME ; 表格中的下一个人名
DEC NAMECTR ; 这是最后一个吗?
JNZ F20 ; 否, 重复
RET ; 是, 退出
F10NAMES ENDP
;
; ----- 公用显示例行程序: -----
G10DISPLY PROC NEAR ; 在入口设置 BP, CX
MOV AX, 1301H ; 使用 AX, BX, DX
MOV BX, 0016H ; 请求显示
MOV DH, ROW ; 属性
MOV DL, 10 ; 行
INT 10H ; 列
RET
G10DISPLY ENDP
;
; ----- 清除屏幕, 设置属性: -----

```

图 14-6 续

```

;
Q10CLEAR PROC NEAR ;使用 AX, BH, CX, DX
MOV AX, 0600H
MOV BH, 61H ;属性
MOV CX, 00 ;填充屏幕
MOV DX, 184FH
INT 10H
RET
Q10CLEAR ENDP
END A10MAIN

```

图 14-6 续

14.7 地址表

程序可能必须测试许多相关的条件，每个条件要求转移到另一个例行程序。例如，考虑下面测试代码 1, 2, 3, 4 等等的程序片断，假设代码是在名为 CODE 的数据项中。处理代码的一般方法是相继地比较每个代码：

```

CODE DB ? ; 代码的数据项
...
CMP CODE, 1 ; 代码=1?
JE CODE1_RTNE
CMP CODE, 2 ; 代码=2?
JE CODE2_RTNE
CMP CODE, 3 ; 代码=3?
JE CODE3_RTNE
...

```

采用这种方法，出错的可能性很大，因为需要与正确的代码去匹配，并转移到正确的例行程序。较为巧妙的解决方案需要一个如下所示的地址表：

```

ADDRTBL DW CODE1_RTNE ; 地址
        DW CODE2_RTNE ; 表
        DW CODE3_RTNE ;
        DW CODE4_RTNE ;
        DW CODE5_RTNE ;
        ...
MOVZX BX, CODE ; 代码进入变址寄存器
DEC BX ; 减 1
SHL BX, 1 ; 并加倍
JMP [ADDRTBL+BX] ; 用地址表转移

```

这个例子把代码传送到 BX 并把 BX 减 1。然后该值加倍，使得 0 还是 0，1 变成 2，2 变成 4，以此类推。这个加倍的值提供的是表格内的偏移值：ADDRTBL+0 是第一个地址(对于代码 1)，ADDRTBL+2 是第二个地址(对于代码 2)，ADDRTBL+4 是第三个地址，以此类推。JMP 指令的操作数[ADDRTBL+BX]形成一个基于表格起始地址加上表格内偏移值的间接地址。然后，操作直接转移到相应的例行程序。

这个例子还可以在 JMP 的位置使用 CALL 指令，其中表格地址是过程名：

```
CALL [ADDRTBL+BX] ; CALL 中使用表格地址
```

在这个例子中的一个重要约束是，代码只可以是十六进制值 1 到最大，任何其他值都可能导致灾难性的后果，因此程序应当检查这种可能性。

14.8 二维数组

二维数组由 y 行与 x 列组成，如下面例子所示的 3 行 5 列：

	0	1	2	3	4	列
0						
行 1						
2						

该数组格式是[*row*, *column*]，并包含 $3 \times 5 = 15$ 个单元或元素。在存储器中，每一行数据可以看成是一维数组，即每行依次跟着另一行。但是，把数组看成是二维的是有帮助的，它可以定义成以下这样：

```
DATA_ARRAY DW 3 DUP(5 DUP(?))
```

访问在该数组中的一个元素，比如[2, 3](即 2 行 3 列)，包括下面一些步骤：

1. 行乘以列中的元素数： $2 \times 5 = 10$ 。
2. 加上列： $10 + 3 = 13$

在这种情况下，所需要的元素是第 13 个(其中行与列都从 0 开始)。注意，在表中的元素是字，但寻址是基于字节的，所以对于行与列的值分别加倍为 20 与 6 是必要的。第 13 个元素是在第 26 个字节并可以用基址：变址寻址进行访问，就像下面这样：

```
MOV BX, 20          ; 行是基址
MOV DI, 6            ; 列是变址
ADD DATA_ARRAY[BX+DI], AX ; 加到数组
```

但是，编一个程序时，难得知道确切的元素地址，因为行与列的值通常是计算得来的结果。

某些时候，把数组看成一维的是比较简单的，如下面例子中要把 DATA_ARRAY 的每个元素都置成 0：

```
MOV CX, 15          ; 元素数
MOV BX, 0            ; 第一个元素的偏移值
L10: MOV DATA_ARRAY[BX], 0 ; 清除一个元素
      ADD BX, 2        ; 数组中的下一个字
      LOOP L10         ; 重复 15 次
```

展开表运算

这一节涉及的是实现展开表(Spreadsheet)运算的技术。在简单的术语中，展开表是由水平与垂直单元组成的二维数组。行的单元总数在右边(最后一列)，列的单元总数横穿过底边(最后一行)。

图 14-7 的程序定义一个展开表，并实现单元的水平相加与垂直相加。为了便于说明，该表格有意小一些并包含预先定义的值。表格包含 5 行的双字，每行有 6 列，第 5 行与第 6 行是总单元数。

程序完成行的水平相加，首先把一行中的每个单元从左到右相加，加到行的总数中。进到下一行并重复水平相加，直到所有行都相加完为止。垂直相加遵循同样的方法，可以通过对指令及其注释的分析，更好地理解这种方法。水平相加可以把每行作为一维来处理，而垂直相加可以把每一列作为一维来处理。

```

TITLE      A14SPRED (EXE) 展开表运算
.MODEL SMALL
.STACK 64
.DATA
SPRD_TBL DW 05, 03, 06, 04, 10, 00 ; 数据行
          DW 04, 05, 01, 09, 06, 00
          DW 06, 02, 00, 10, 11, 00
          DW 10, 07, 03, 05, 02, 00
          DW 00, 00, 00, 00, 00, 00 ; 总计行

COUNT DB 7
COLS EQU 12 ; 6列 × 2字节
ROWS EQU 10 ; 5行 × 2字节
NOCOLS EQU 06 ; 列数
NOROWS EQU 05 ; 行数
COLTOT EQU NOCOLS*(NOROWS-1)*2 ; 6列 × 4行 × 2
; -----
.386

A10MAIN PROC FAR
MOV AX,@data ; 初始化
MOV DS,AX ; 段
MOV ES,AX ; 寄存器
CALL B10HORZL ; 调用水平加法
CALL C10VERTL ; 调用垂直加法
;
MOV AX,4C00H ; 结束处理
INT 21H
A10MAIN ENDP

;
; ----- 加水平总数 (行) : -----
B10HORZL PROC NEAR
PUSHA ; 保存寄存器
MOV COUNT,NOROWS-1 ; 行的计数
LEA DI,SPRD_TBL ; 初始化表格
LEA SI,SPRD_TBL ; 地址
ADD SI,ROWS
B10: MOV CX,NOCOLS-1 ; 列的计数
B20: MOV AX,[DI] ; 从单元中取数
ADD [SI],AX ; 加到行的总数中
ADD DI,02 ; 行中的下一单元
LOOP B20 ; 重复通过行的所有单元
ADD DI,02 ; 下一行的第一个单元
ADD SI,COLS ; 下一行总数
DEC COUNT ; 重复通过所有行
JNZ B10
POPA ; 恢复寄存器
RET
B10HORZL ENDP
;
; ----- 加垂直总数 (列) : -----
C10VERTL PROC NEAR
PUSHA ; 保存寄存器
MOV COUNT,NOCOLS-1 ; 列的计数
MOV BX,00
LEA SI,[SPRD_TBL+COLTOT] ; 初始化列总数
C10:

```

图 14-7 展开表运算

```

                LEA    DI,SPRD_TBL      ;初始化顶部单元
                ADD    DI,BX
                MOV    CX,NROWS-1      ;行的计数
C20:            MOV    AX,[DI]          ;从单元中取数
                ADD    [SI],AX         ;加到总数中
                ADD    DI,COLS        ;列中下一单元
                LOOP   C20             ;重复通过列中所有单元
                ADD    BX,02          ;行中下一单元
                ADD    SI,02          ;下一列总数
                DEC    COUNT          ;重复通过所有列
                JNZ    C10            ;恢复寄存器
                POPA
                RET
C10VERTL      ENDP
                END    A10MAIN

```

图 14-7 续

14.9 要 点

- 对于大多数用途来说，表格所含的相关项目具有相同的长度和数据格式。
- 表格是建立在它的数据格式基础之上的，例如，项目可能是字符或数字，典型的是每个项目长度一样。
- 如果表格经常改变，或者几个程序引用该表格，那么表格可以放在磁盘上。一个作修改的程序可以处理对表格的修改。因此，任何程序都可以从磁盘取表格，而程序并不需要改变。
- 在直接表格寻址条件下，程序计算表格项目的地址并直接访问该项目。
- 当查找一个表格时，程序要把一个数据项和表格中的每个项目相继地进行比较，直到发现符合(找到)为止。CMP 和 CMPSW 认定字所包含的字节是按相反顺序排列的。
- XLAT 指令便于把数据从一种格式转换成另一种格式。

14.10 习 题

14-1. 说明用直接寻址与用查找处理表格之间的区别。

14-2. 定义一个名为 ANNUAL_TBL 有 365 个字的表格，初始化为(a)对字符数据是空格，(b)对二进制数据是零。

14-3. 分别定义 3 个有联系的表格，它们包含以下数据：(a)ASCII 的项目编号 04, 07, 14, 17, 以及 24, (b)项目说明：DVDs, receivers, modems, keyboards, 以及 diskettes, (c)项目的价格：22.20, 95.75, 47.45, 49.35, 以及 12.95。

14-4. 修改 14-3 题，使所有数据都在同一张表中。对于第一个项目，把它的编号和说明定义在第 1 行中，而它的价格定义在第 2 行中；对于第二个项目，把它们定义在第 3 行和第 4 行，以此类推。

14-5. 修改图 14-1，使之采用数字(ASCII)格式接受来自键盘的月份。如果输入是有效的

(01-12), 定位并显示字母表示的月份; 否则, 显示出错信息。允许任何数量的键盘输入, 当用户只用回车键回答提示符时, 结束处理。

14-6. 编一个程序, 允许用户从键盘输入项目编号和数量。使用 14-4 题定义的表格并包括一个查找例行程序, 该程序用输入项目编号把它定位在表格中。从表中提取说明与价格。计算每次销售的值(数量 \times 价格), 并在屏幕上显示说明和值。当用户只用回车键回答提示符时, 结束处理。

14-7. 使用 14-3 题定义的说明表, 编写一个程序(a)把该表的内容传送到另外一个(空的)表, (b)将这一新表的内容按升序排列, (c)按屏幕上相连续的行显示每个说明。提供卷动的屏幕。

14-8. 在“XLAT 指令”一节中的程序片断把 ASCII 字符转换成 EBCDIC 格式。修改该例子, 进行相反的处理——把 EBCDIC 数据转换成 ASCII 格式。EBCDIC 字符要转换的是负的符号(60H), 十进制小数点(48H), 数 0-9(F0H-F9H), 以及所有其他字符转换成空格。对于数据, 使用 EBCDIC 的十六进制字符串 F0F0F1F24BF5F060(定义成 0F0H, 0F1H 等等), 把它转换成 ASCII 格式并加以显示。该十六进制结果应当是 303031322E35302D。

14-9. 编写一个程序, 为数据提供简单的加密。定义一个名为 CRYPTDATA 的 80 字节的数据区, 它包含任何 ASCII 数据。安排一个转换表把数据转换成有点随机的形式, 如 A 变成 M, B 变成 R, C 变成 X, 如此等等。提供全部 256 种可能的字节值。安排第二个转换表, 按相反的方式(解密)处理该数据。程序应当完成以下动作: (a)在一行中显示 CRYPTDATA 的原始内容, (b)加密 CRYPTDATA 并在第二行显示加密后数据, (c)解密 CRPYTDATA 并在第三行显示解密后的数据(它应该和第一行内容相同)。

14-10. 修改图 14-7 中的程序(二维表格), 使之把每行的右边的总数相加(垂直地), 并把每列的底部的总数相加(水平地)。

第五部分

高級輸入/輸出

目的：描述使用鼠标的程序设计要求

15.1 引言

这一章描述了鼠标的使用：初始化鼠标，显示、隐藏鼠标指针，设置指针的位置和范围，和获取鼠标按键的信息。两个程序举例说明了鼠标处理的使用。鼠标处理只介绍一条新指令 INT 33H。

鼠标是一个通用的指示设备，它由一个统称为设备驱动程序的软件接口来控制，这个驱动程序一般由 CONFIG.SYS 或 AUTOEXEC.BAT 文件的一个项来安装。必须安装这个驱动程序，这样一个程序才可以识别并响应鼠标的动作。

下面是一些基本的鼠标定义：

- 像素：屏幕上最小的可寻址的元素。例如，对文本方式 03，每字节有 8 个像素。
- 鼠标指针：在文本方式下，指针是一个反相显示的闪烁的方块；在图形方式下，指针是一个箭头。
- Mickey：测量鼠标移动的单位，约等于 1/200 英寸。
- Mickey 计数：鼠标球横向或纵向滚动的 mickey 的数量。鼠标驱动器用 mickey 来计算指针在屏幕上移动的像素数。
- 阈值速度：阈值速度以每秒 mickey 为单位，鼠标必须在屏幕上以双倍速度移动指针。默认的阈值速度是每秒 64 mickeys。

程序中的所有鼠标操作都是由标准的 INT 33H 功能来执行的，其格式为：

MOV	AX, function	: 请求鼠标功能
...		: 参数 (如果有的话)
INT	33H	: 调用鼠标驱动程序

注意，不像其他的 INT 操作使用 AH 寄存器，INT 33H 的功能号被装入整个的 AX 寄存器。

程序发出的第一条鼠标指令应当是功能 00H，它仅仅初始化鼠标驱动器和程序之间的接口。典型的用法是，在程序开始只需发出一次这个命令。紧接着功能 00H，程序应当执行功能 01H，它使鼠标指针出现在屏幕上。随后，可以在鼠标操作的范围内选择其他功能。

以下是可用到的 INT 33H 的鼠标功能，其中有一些是经常要用到的：

00H 初始化鼠标

01H	显示鼠标指针
02H	隐藏鼠标指针
03H	获取按键状态和指针位置
04H	设置指针位置
05H	获取点击按键信息
06H	获取释放按键信息
07H	设置指针水平界限
08H	设置指针垂直界限
09H	设置图形指针类型
0AH	设置文本指针类型
0BH	读取鼠标移动计数器
0CH	为鼠标事件安装中断处理程序
0DH	打开光笔仿真
0EH	关闭光笔仿真
0FH	设置 mickey 与像素的比率
10H	设置指针禁止区
13H	设置倍速阈值
14H	调动鼠标事件中断
15H	获取鼠标驱动器状态的缓冲区大小
16H	保存鼠标驱动器状态
17H	恢复鼠标驱动器状态
18H	安装鼠标事件的可选处理器
19H	获取可选处理器的地址
1AH	设置鼠标灵敏度
1BH	获取鼠标灵敏度
1CH	设置鼠标中断速率
1DH	选择指针的显示页
1EH	获取指针的显示页
1FH	禁止鼠标驱动器
20H	允许鼠标驱动器
21H	重置鼠标驱动器
22H	设置鼠标驱动器信息的语种
23H	获取语种数
24H	获取鼠标信息

15.2 基本的鼠标操作

以下几节描述了使用鼠标的程序所需要的基本 INT 33H 操作。

1. 功能 00H: 初始化鼠标。这是一个程序为处理一个鼠标并只需执行一次的第一个命令。功能 00H 装入 AX, 不带其他的输入参数, 然后执行 INT 33H。此操作返回这些值:

- AX=0000H, 如果没有可用的鼠标支持程序; 或 AX=FFFFH, 如果支持程序是可用的
- BX=鼠标按键的数量, 如果支持程序是可用的

如果鼠标支持程序是可用的, 该操作初始化鼠标的步骤如下:

- 设置鼠标指针到屏幕中央
- 如果鼠标指针是可见的, 则隐蔽鼠标指针
- 设置鼠标指针的显示页为零
- 根据屏幕显示方式设置鼠标指针: 文本方式为矩形及反相彩色, 图形方式为箭头
- 设置 mickey 与像素的比率, 水平比率=8:8, 垂直比率=16:8
- 设置指针的水平和垂直界限的最小值和最大值
- 设置倍速阈值为每秒 64 mickeys, 该值可以改变。

2. 功能 01H: 显示鼠标指针。这个操作在功能 00H 后使用, 它使鼠标指针显示在屏幕上。这个操作不需要输入参数, 也没有返回值。

鼠标驱动器保存一个指针标志以确定是否显示指针。如果标志是 0 则显示指针, 是其他值则隐蔽指针。这个值初始化为-1, 功能 01H 递增这个标志为 0, 这样指针就被显示。(参见功能 02H。)

3. 功能 02H: 隐蔽鼠标指针。标准的做法是在程序执行的最后调用这个功能以隐蔽指针。这个操作不需要输入参数, 也不返回任何值。

当指针标志包含一个 0 时显示指针, 当为其他值时隐蔽指针。这个功能递减指针标志从 0 到-1, 使指针被隐蔽。

4. 功能 03H: 获取按键状态和指针位置。这个功能不需要输入参数, 返回有关鼠标的信息如下:

- BX=按键的状态, 根据位的位置确定如下:

Bit 0 左键 (0=未按, 1=按下)

Bit 1 右键 (0=未按, 1=按下)

Bit 2 中键 (0=未按, 1=按下)

Bit 3 至 15 为保留内部使用

- CX=水平 (x) 坐标

- DX=垂直 (y) 坐标

水平和垂直坐标在像素项中表示, 即使在文本方式下也这样(显示方式 03 为每字节 8 个像素)。这些值通常在指针的最小值和最大值界限之内。

5. 功能 04H: 设置指针位置。这个操作设置鼠标在屏幕上的水平和垂直坐标 (位置的值在像素项中——对显示方式 03 为每字节 8 个像素):

```
MOV     AX, 04H           ; 请求设置鼠标指针
MOV     CX, horizontal    ; 水平位置
MOV     DX, vertical      ; 垂直位置
INT     33H               ; 调用鼠标驱动程序
```

如果指针超出了最小和最大的界限, 此操作会做出必要的调整, 在新位置设置指针。

15.3 程序：显示鼠标位置

图 15-1 的程序说明了基本鼠标操作。在用户移动鼠标，而不是按鼠标的时候，它显示指针的水平和垂直坐标。主要过程如下：

- A10MAIN 初始化程序，调用 B10INITZ，C10POINTR，D10CONVRT 和 E10DISPLY。当用户按动左键，程序使用功能 02H 隐蔽指针并结束处理。

- B10INITZ 调用 INT 33H 的功能 00H 初始化鼠标（或指示没有鼠标驱动程序），并且调用功能 01H 显示鼠标指针。

- 如果用户按动了左键，C10POINTR 调用功能 03H 检测并退出。如果没有按键，程序将水平和垂直坐标从像素值转换为二进制数（把这个值右移 3 位，相当于除 8）。如果位置和先前检测到的一样，例行程序重复调用功能 03H；如果位置变化了，控制返回到调用程序。

- D10CONVRT 将水平和垂直的屏幕位置的二进制值转换为可显示的 ASCII 字符。注意每字节有 8 个像素，在屏幕上第 79 列（最右边的位置）返回的水平值是 $79 \times 8 = 632$ 。程序把水平值除以 8 以得到最大值，在这个例子中是 79。因此，这个转换确保返回值是在 0 到 79 之内。

- E10DISPLY 在屏幕中心显示水平与垂直坐标，如 X=col, Y=row。

```

TITLE      A15MOUSE (EXE)  Handling the mouse
.MODEL     SMALL
.STACK     64
.DATA
LEN_DATA   EQU      14
XCOORD     DW        0           ; 显示长度
YCOORD     DW        0           ; 二进制的 X 坐标
ASCVAL     DW        ?           ; 二进制的 Y 坐标
                                ; ASCII 域

DISPDATA   LABEL    BYTE
XMSG       DB        'X = '      ; 屏幕显示域:
XASCII     DW        ?           ; X 的信息
                                ; X 的 ASCII 值
                                ;
YMSG       DB        'Y = '      ; Y 的信息
YASCII     DW        ?           ; Y 的 ASCII 值
.386
;-----
.CODE
A10MAIN    PROC      FAR
MOV        AX,@data           ; 初始化
MOV        DS,AX              ; DS 和 ES
MOV        ES,AX              ; 地址
CALL       Q10CLEAR           ; 清屏
CALL       B10INITZ           ; 初始化鼠标
CMP        AX,00               ; 已安装鼠标?
JE         A90                 ; 否, 退出
A20:       CALL      C10POINTR  ; 取鼠标指针
CMP        BX,01               ; 按下键?
JE         A80                 ; 是, 退出
MOV        AX,XCOORD           ; X 转换为
CALL       D10CONVRT           ; ASCII

```

图 15-1 使用鼠标

```

MOV     AX,ASCVAL      ;
MOV     XASCII,AX      ;
MOV     AX,YCOORD      ; Y转换为
CALL    D10CONVRT      ; ASCII
MOV     AX,ASCVAL      ;
MOV     YASCII,AX      ; 显示
CALL    E10DISPLY      ; X和Y
JMP     A20            ; 重复
A80:    MOV     AX,02H   ; 请求隐藏指针
        INT     33H
A90:    CALL    Q10CLEAR ; 清屏
        MOV     AX,4C00H ; 处理结束
        INT     21H
A10MAIN ENDP
;
;
B10INITZ PROC NEAR      ; 使用 AX
MOV     AX,00H          ; 请求初始化
INT     33H             ; 鼠标
CMP     AX,00           ; 鼠标已安装?
JE      B90             ; 否,退出
MOV     AX,01H          ; 显示鼠标指针
INT     33H
B90:    RET            ; 返回调用程序
B10INITZ ENDP

;
; 获取鼠标指针位置:
C10POINTR PROC NEAR      ; 使用 AX, BX, CX, DX
C20:    MOV     AX,03H   ; 取指针位置
        INT     33H
        CMP     BX,00000001B ; 左键按下?
        JE      C90      ; 是,表示退出
        SHR     CX,03     ; 像素坐标
        SHR     DX,03     ; 除8
        CMP     CX,XCOORD ; 指针位置
        JNE     C30      ; 改变?
        CMP     DX,YCOORD ;
        JE      C20      ; 否,重复操作
C30:    MOV     XCOORD,CX ; 是,保存新位置
        MOV     YCOORD,DX ;
C90:    RET            ; 返回调用程序
C10POINTR ENDP

;
; 二进制的 X 或 Y 位置转换为 ASCII:
;AX 已设置输入数据 = 二进制的 X 或 Y
D10CONVRT PROC NEAR      ; 使用 CX, SI
MOV     ASCVAL,2020H     ; 清除 ASCII 域
MOV     CX,10            ; 设置除数
LEA     SI,ASCVAL+1      ; 装入 ASCVAL 地址
CMP     AX,CX            ; 位置与 10 比较
JB      D20              ; 低于10,跳转
DIV     CL                ; 高于10,除以10
OR      AH,30H           ; 转换成 ASCII
MOV     [SI],AH          ; 存入最右边字节
DEC     SI                ; ASCVAL 地址减1
D20:    OR      AL,30H    ; 变为 ASCII
        MOV     [SI],AL  ; 存入最左边字节
        RET            ; 返回调用程序
D10CONVRT ENDP

;
; 显示 X, Y 位置
E10DISPLY PROC NEAR      ; 使用 AX, BX, BP, CX, DX
MOV     AX,1300H         ; 请求显示
MOV     BX,0031H         ; 页:属性
LEA     BP,DISPDATA      ; 串地址
MOV     CX,LEN_DATA      ; 字符数
MOV     DX,0020H         ; 屏幕行:列
INT     10H
RET
E10DISPLY ENDP

```

图 15-1 续

```

;
;
;-----清屏, 设置属性-----
Q10CLEAR PROC NEAR ;使用 AX, BX, BP, CX, DX
MOV AX, 0600H ;请求清屏
MOV BH, 30H ;颜色
MOV CX, 00 ;全屏
MOV DX, 184FH ;
INT 10H
RET ;返回调用程序
Q10CLEAR ENDP
END A10MAIN

```

图 15-1 续

改进这个程序的一个方法是调用功能 0CH 来设置中断处理程序。用这种方法, 只要鼠标是激活的, 在鼠标中断处理程序中自动包括了所需要的指令。

15.4 更高级的鼠标操作

这一节包含了其余的鼠标操作, 下一节提供了另一个程序例子。

1. 功能 05H: 获取点击按键信息。这个功能返回点击按键的信息。设置 BX 为按键编号, 0=左键, 1=右键, 2=中键:

```

MOV AX, 05H ; 请求按键信息
MOV BX, button_no ; 按键号
INT 33H ; 调用鼠标驱动程序

```

这个操作返回按键的未按/按下的状态、点击次数和所请求按键的位置:

- AX=按键的状态, 根据位的位置确定键, 如下:

Bit 0 左键 (0=未按, 1=按下)

Bit 1 右键 (0=未按, 1=按下)

Bit 2 中键 (0=未按, 1=按下)

Bit 3-15 保留内部使用

- BX=点击按键计数器
- CX=最后一次点击的水平 (x) 坐标 (像素值)
- DX=最后一次点击的垂直 (y) 坐标 (像素值)

该操作重置点击按键计数器为 0。

2. 功能 06H: 获取释放键信息。这个功能返回按键释放的信息。设置 BX 按键号 (0=左键, 1=右键, 2=中键):

```

MOV AX, 06H ; 请求释放键信息
MOV BX, button_no ; 按键号
INT 33H ; 调用鼠标驱动程序

```

这个操作返回所有按键的未按/按下的状态、释放次数和所请求的按键的位置:

- AX=按键的状态, 根据位的位置确定键, 如下:

Bit 0 左键 (0=未按, 1=按下)

Bit 1 右键 (0=未按, 1=按下)

Bit 2 中键 (0=未按, 1=按下)

Bit 3—15 保留内部使用

- BX=释放键计数器
- CX=最后一次释放键的水平 (x) 坐标 (像素值)
- DX=最后一次释放键的垂直 (y) 坐标 (像素值)

操作重置释放键计数器为 0。

3. 功能 07H: 设置指针水平界限。这个操作设置指针的最小和最大的水平界限 (像素值):

MOV	AX, 07H	; 请求设置水平界限
MOV	CX, minimum	; 最小界限
MOV	DX, maximum	; 最大界限
INT	33H	; 调用鼠标驱动程序

如果最小值大于最大值, 该操作随意交换这两个值。如果指针超出了定义的区域, 该操作把它移回到区域内。参见功能 08H 和 10H。

4. 功能 08H: 设置指针垂直界限。这个操作设置指针的最小和最大的垂直界限 (像素值):

MOV	AX, 08H	; 请求设置垂直界限
MOV	CX, minimum	; 最小界限
MOV	DX, maximum	; 最大界限
INT	33H	; 调用鼠标驱动程序

如果最小值大于最大值, 该操作任意交换这两个值。如果指针超出了定义区域, 该操作把它移回到区域内。参见功能 07H 和 10H。

5. 功能 0BH: 读取鼠标移动计数器。这个操作返回自最后一次请求这个功能以来的水平和垂直的 mickey 数 (在 -32768 到 +32767 范围内)。返回值是:

- CX=水平计数 (一个正值表示移动到右边, 负值表示移动到左边)
- DX=垂直计数 (一个正值表示向下移动, 负值表示向上移动)

6. 功能 0CH: 安装鼠标事件的中断处理程序。程序可能需要在鼠标发生关联的动作 (或事件) 时自动响应。功能 0CH 的目的是提供一个事件处理程序, 借此鼠标软件来中断程序并调用事件处理程序, 执行它所请求的功能并在完成任务的时候返回到程序的执行点 (断点)。

CX 装入一个事件屏蔽码, 以指出处理程序响应哪一种事件, ES: DX 装入中断处理程序的段地址: 偏移地址:

MOV	AX, 0CH	; 请求中断处理程序
LEA	CX, mask	; 事件屏蔽码地址
LEA	DX, handler	; 处理程序地址 (ES: DX)
INT	33H	; 调用鼠标驱动程序

定义事件屏蔽码的各位设置要求如下:

0=移动的鼠标指针	4=释放右键
1=点击左键	5=点击中键
2=释放左键	6=释放中键
3=点击右键	7-15=保留, 定义为 0

定义中断处理程序为一个 FAR 过程。鼠标驱动程序使用一个远程调用进入中断处理程序, 并设置如下寄存器:

- AX=定义的事件屏蔽码，只有在条件发生时才设置的那些位
- BX=按键状态（如果已设置，位0代表左键按下，位1代表右键按下，位2代表中键按下）

- CX=水平坐标（x）
- DX=垂直坐标（y）
- SI=最后的垂直 mickey 数
- DI=最后的水平 mickey 数
- DS=鼠标驱动程序的数据段

在进入中断处理程序的入口处，把所有的寄存器都入栈保存，并将 DS 初始化为数据段的地址。在处理程序中，只能使用 BIOS 中断，而不能用 DOS 中断。退出时，所有寄存器都出栈。

7. 功能 10H: 设置指针禁止区域。这个操作定义了一个不显示指针的屏幕区域:

```
MOV     AX, 10H           ; 请求设置禁区
MOV     CX, upleft_x      ; 左上角 x 坐标
MOV     DX, upleft_y      ; 左上角 y 坐标
MOV     SI, lowright_x    ; 右下角 x 坐标
MOV     DI, lowright_y    ; 右下角 y 坐标
INT     33H               ; 调用鼠标驱动程序
```

用不同的参数重新调用这个函数，或重新调用函数 00H 或 01H 来替换禁区。

8. 功能 13H: 设置倍速阈值。这个操作设置阈值速度，指针以双倍于它的速度在屏幕上移动。在 DX 中装入新的值（默认值是每秒 64 mickey）。（参见功能 1AH）

9. 功能 1AH: 设置鼠标灵敏度。灵敏度与指针移动之前鼠标必须移动的 mickey 数有关。这个功能按照每 8 个像素的 mickey 数来设置水平的和垂直的鼠标移动量，以及使指针以双倍的速度在屏幕上移动的阈值速度，（参见功能 0FH, 13H 和 1BH）:

MOV	AX, 1AH	; 请求设置鼠标灵敏度
MOV	BX, horizontal	; 水平 mickey 数（默认值=8）
MOV	CX, Vertical	; 垂直 mickey 数（默认值=16）
MOV	DX, threshold	; 阈值速度（默认值=64）
INT	33H	; 调用鼠标驱动程序

10. 功能 1BH: 获取鼠标灵敏度。这个操作根据每 8 个像素的 mickey 数来返回水平的和垂直的鼠标移动量，以及阈值速度，指针在屏幕上以其双倍的速度来移动。（参见功能 1AH, 返回的寄存器及相应值）

11. 功能 1DH: 选择指针的显示页。视频显示页用 INT 10H 的功能 05H 设置。对于鼠标操作，在 BX 中设置页号，并调用这个函数。

12. 功能 1EH: 获取指针的显示页。这个操作在 BX 中返回当前的视频显示页。

13. 功能 24H: 获取鼠标信息。这个操作返回有关已安装鼠标的版本和类型信息:

BH=主版本号

BL=子版本号

CH=鼠标类型（1=总线鼠标，2=串口鼠标）

15.5 程序：按菜单使用鼠标

在前面图 10-2 的程序使用了光标键从菜单里选择一项。图 15-2 的程序是类似的，但是现在允许用户在菜单上上下下移动鼠标指针，并通过点击左键选中一个入口项。同样的，现在在菜单最下面有一项是“Exit Program”。主要的过程如下：

- A10MAIN 调用 B10INITZ 初始化鼠标，调用 C10MENU 显示菜单，调用 E10DISPLY 来高亮度显示当前菜单项，调用 D10POINTR 响应鼠标的动作，并且当用户请求“Exit Program”时，结束处理。

- B10INITZ 初始化鼠标，显示指针，并且为指针区域设置水平和垂直界限。

- C10MENU 显示全部菜单选项。

- D10POINTR 检验是否点击左键，如果点击，调用 E10DISPLY 设置原先的菜单项为正常显示并把选择项设置成反相显示。

- E10DISPLY 按照给定的属性显示菜单项。

```

TITLE      A15SELMU (EXE) 选择菜单项
.MODEL     SMALL
.STACK     54
.DATA
TOPROW     EQU      08           ; 菜单上边行
BOTROW     EQU      16           ; 菜单底边行
LEFTCOL    EQU      26           ; 菜单左边列
LEN_LINE   EQU      19           ; 菜单行的长度
ATTRIB     DB        ?           ; 屏幕属性
COL        DB        00           ; 屏幕列
ROW        DB        00           ; 屏幕行
SHADOW     DB        19 DUP(0DBH) ; 阴影符
MENU       DB        0C9H, 17 DUP(0CDH), 0BBH
           DB        0BAH, ' Add records ', 0BAH
           DB        0BAH, ' Delete records ', 0BAH
           DB        0BAH, ' Enter orders ', 0BAH
           DB        0BAH, ' Print report ', 0BAH
           DB        0BAH, ' Update accounts ', 0BAH
           DB        0BAH, ' View records ', 0BAH
           DB        0BAH, ' Exit program ', 0BAH
           DB        0C8H, 17 DUP(0CDH), 0BCH
PROMPT     DB        'To select an item, press left '
           DB        'button of mouse pointer.'
.386 ; -----
.CODE
A10MAIN    PROC     FAR
MOV        AX,@data           ; 初始化段
MOV        DS,AX              ; 寄存器
MOV        ES,AX
CALL       Q10CLEAR            ; 清屏
CALL       B10INITZ            ; 初始化鼠标
CMP        AX,00               ; 已安装鼠标?
JE         A90                 ; 否，退出
CALL       C10MENU             ; 显示菜单
A20:
MOV        ROW,TOPROW+1        ; 设置上部菜单项的行
MOV        ATTRIB,16H          ; 设置为反相显示
CALL       E10DISPLY           ; 当前菜单行为高亮度
CALL       D10POINTR           ; 调用鼠标例程
CMP        DX,BOTROW-1        ; 请求退出?
JNE        A20                 ; 否，继续

```

图 15-2 选择菜单

```

                MOV     AX,02H           ; 隐藏鼠标指针
                INT     33H
                MOV     AX,0600H         ; 清屏
                CALL    Q10CLEAR
                ;
A90:            MOV     AX,4C00H         ; 结束处理
                INT     21H
A10MAIN        ENDP
;
;          初始化鼠标指针, 设置水平和垂直界限:
;
B10INITZ      PROC    NEAR             ; 使用 AX, CX, DX
                MOV     AX,00H           ; 请求初始化
                INT     33H             ; 鼠标
                CMP     AX,00           ; 已安装鼠标?
                JE      B90             ; 否, 退出

                MOV     AX,01H           ; 显示指针
                INT     33H
                MOV     AX,04H           ; 设置指针
                MOV     CX,256
                MOV     DX,108
                INT     33H
                MOV     AX,07H           ; 水平界限
                MOV     CX,LEFTCOL+1     ; 左边列
                MOV     DX,LEFTCOL+17    ; 右边列
                SHL     CX,03            ; 像素值
                SHL     DX,03            ; 乘 8
                INT     33H
                MOV     AX,08H           ; 垂直界限
                MOV     CX,TOPROW+1      ; 上边行
                MOV     DX,BOTROW-1      ; 底边行
                SHL     CX,03            ; 除 8
                SHL     DX,03
                INT     33H
B90:            RET
B10INITZ      ENDP
;
;          显示阴影框和全部菜单:
;
C10MENU       PROC    NEAR             ; 使用 AX, BP, BX, CX, DX
                MOV     AX,1301H         ; 请求显示
                MOV     BX,0060H         ; 棕底黑字
                LEA     BP,SHADOW        ; 阴影符的地址
                MOV     CX,LEN_LINE      ; 行的长度
                MOV     DH,TOPROW+1      ; 屏幕行
                MOV     DL,LEFTCOL+1     ; 和列
C20:            INT     10H
                INC     DH               ; 下一行
                CMP     DH,BOTROW+2      ; 全部行显示完?
                JNE     C20              ; 否, 重复
                MOV     ATTRIB,71H       ; 白底蓝色
                MOV     AX,1300H         ; 请求显示
                MOV     BH,00            ; 0 页
                MOV     BL,ATTRIB        ; 属性
                LEA     BP,MENU          ; 菜单地址
                MOV     CX,LEN_LINE      ; 行的长度
                MOV     DH,TOPROW        ; 屏幕行,
                MOV     DL,LEFTCOL        ; 列
C30:            INT     10H
                ADD     BP,LEN_LINE      ; 下一个菜单行
                INC     DH               ; 下一行
                CMP     DH,BOTROW+1      ; 全部行显示完?
                JNE     C30              ; 否, 重复
                MOV     AX,1300H         ; 请求显示
                MOV     BH,00            ; 0 页
                MOV     BL,ATTRIB        ; 属性
                LEA     BP,PROMPT        ; 提示行
                MOV     CX,45            ; 提示符长度
                MOV     DH,BOTROW+4      ; 屏幕行,
                MOV     DL,15            ; 列
                INT     10H
                RET
C10MENU       ENDP

```

图 15-2 续

```

;                                     如果左键按下，设置原来的菜单行为正常显示，
;                                     新的行为反相显示：
;
D10POINTR PROC NEAR                                ; 使用 AX, BX, DX
D20:  MOV     AX, 03H                                ; 取按键状态
      INT     33H
      CMP     BX, 00000001B                          ; 左键按下?
      JNE     D20                                    ; 否，重复
      SHR     DX, 03                                  ; 纵向除 8
      CMP     DX, BOTROW-1                            ; 请求退出?
      JE      D90                                     ; 是，退出
      PUSH    DX                                       ; 否，保存行
      MOV     ATTRIB, 71H                             ; 白底蓝字
      CALL    E10DISPLY                               ; 设置原来行为正常显示
      POP     DX                                       ; 取行
      MOV     ROW, DL
      MOV     ATTRIB, 17H                             ; 蓝底白字
      CALL    E10DISPLY                               ; 设置新行为反相显示
      JMP     D20                                     ; 重复
D90:  RET
D10POINTR ENDP

;                                     设置菜单行为正常或高亮度：
;
E10DISPLY PROC NEAR                                ; 使用 AX, BX, BP, CX, DX
MOVZX  AX, ROW                                       ; ROW 说明是哪行菜单行
SUB     AX, TOPROW
IMUL    AX, LEN_LINE                               ; 乘以行长度
LEA     SI, MENU+1                                  ; 选择菜单行
ADD     SI, AX
MOV     AX, 1300H                                    ; 请求显示
MOV     BH, 00                                       ; 页
MOV     BL, ATTRIB                                   ; 新属性
MOV     BP, SI                                       ; 菜单行
MOV     CX, LEN_LINE-2                             ; 串长度
MOV     DH, ROW                                     ; 行
MOV     DL, LEFTCOL+1                               ; 列
      INT     10H
      RET
E10DISPLY ENDP

;                                     清屏，设置属性：
;
Q10CLEAR PROC NEAR                                ; 使用 AX, BH, CX, DX
MOV     AX, 0600H                                    ; 请求清屏
MOV     BH, 61H                                       ; 棕底蓝字
MOV     CX, 0000                                       ; 全屏
MOV     DX, 184FH
      INT     10H
      RET
Q10CLEAR ENDP
END A10MAIN

```

图 15-2 续

15.6 要 点

- 在文本方式下，鼠标指针为反相显示的闪烁方块；在图形方式下，指针为一个箭头。
- 鼠标操作使用 INT 33H，功能码装在 AX 中。
- 执行的第一个鼠标操作应当是 INT 33H 的功能 00H，它初始化鼠标驱动程序。
- 请求 INT 33H 的功能 01H 显示鼠标指针，功能 03H 获得按键状态，功能 04H 获取指针位置，功能 05H 获取按键点击信息，功能 06H 获取释放键信息。
- 鼠标位置的水平和垂直坐标用像素值来表示。

15.7 习 题

15-1. 解释术语: (a) mickey, (b) mickey 计数, (c) 鼠标指针。

15-2. 为下列每个鼠标操作提供 INT 33H 功能:

- (a) 隐蔽鼠标指针
- (b) 获取按键点击信息
- (c) 设置指针位置
- (d) 安装鼠标事件中断处理程序
- (e) 获取释放键信息
- (f) 读取鼠标移动计数器

15-3. 解释鼠标指针标志的用途。

15-4. 按下列要求编写指令:

- (a) 初始化鼠标
- (b) 显示鼠标指针
- (c) 获取鼠标信息
- (d) 在中间一列的第 22 行设置鼠标指针
- (e) 获取鼠标灵敏度
- (f) 获取按键状态和指针位置
- (g) 隐蔽鼠标指针

15-5. 结合问题 15-4 中的要求, 编写完整的程序。并在 DEBUG 下运行程序, 虽然有时 DEBUG 会使指针滚动出屏幕。

15-6. 编写指令, 设置指针禁区为(a)左上角: $x=40$, $y=40$, (b)右下角: $x=160$, $y=80$ 。

磁盘存储 I: 组织方式

目的: 分析硬盘及软盘存储的基本格式、引导记录、目录和文件分配表。

16.1 引言

一个专业的程序员必须熟悉磁盘组织的技术细节, 特别是在开发检测软盘、硬盘和 CD-ROM 内容的实用程序时尤其如此。

本章解释了磁道、扇区和柱面的概念, 并且给出了一些常用设备的容量, 还介绍了磁盘起始区中的重要数据记录的组织, 包括引导记录(帮助把操作系统从磁盘装入到内存)、目录(包含了文件名、位置和每个文件在磁盘上的状态)和文件分配表(或 FAT, 为文件分配磁盘空间)。

本文在需要提及硬磁盘和软磁盘的地方都使用通用术语磁盘。

16.2 磁盘存储设备的特征

为了处理磁盘上的记录, 必须熟悉一些术语和磁盘组织的特征。一个软磁盘有两面(或表面), 而硬盘在一个轴上包含有许多双面磁盘。

16.2.1 磁道和扇区

硬磁盘或软磁盘的每面都包含许多同心的磁道, 从最外面的磁道以 00 开始编号, 每个磁道格式化为 512 字节的可以存储数据的扇区。

软磁盘和硬磁盘设备都由一个控制器控制运转, 控制器处理读写头在磁盘表面上的位置, 以及在磁盘与存储器之间的数据传输。每个磁盘表面都有一个读写头。对于软磁盘和硬磁盘, 一个读或写操作的请求都会导致磁盘驱动控制器把读写头(如必要的话)移动到所要求的磁道上。然后控制器等待旋转面上所要求的扇区到达读写头下面, 这时发生读或写操作。例如, 对于一次读操作, 当扇区经过读写头时, 控制器就读取扇区上的每一位。图 16-1 说明了这些特征。

硬盘和软盘驱动器主要在两点上不同。对于硬盘, 读写头正好悬在磁盘表面上, 不总是

与盘面接触,而对于软盘,读写头实际上一直接触着盘面。另外硬盘设备总在旋转,而软磁盘设备在每次读/写操作时都要启动和停止。

16.2.2 柱面

柱面是每个软盘或硬盘表面上同一编号的磁道在垂直方向上的集合。因此柱面 0 就是每个盘面上所有编号为 0 的磁道的集合,柱面 1 是所有编号为 1 的磁道的集合,依此类推。对于一个软盘来说,柱面 0 由盘面 1 的磁道 0 和盘面 2 的磁道 0 组成;柱面 1 由盘面 1 的磁道 1 和盘面 2 的磁道 1 组成;依此类推。盘面的编号和读写头的编号是相同的,例如,读写头 1 访问盘面 1 上的数据。

当写文件时,控制器在一个柱面的所有磁道上填写,然后把读写头前移到下一个柱面。例如,系统填写软盘柱面 0(盘面 1 和盘面 2 上磁道 0 的所有扇区),然后前移到盘面 1 的柱面 1 上。

可见对盘面(读写头)、磁道和扇区的定位都是通过编号来进行的。盘面和磁道从 0 开始编号,但是扇区可能用下面两种方法之一来编号:

1. 物理分区法,每个磁道上的扇区从 1 开始编号,这样磁盘上的第一个扇区就编址为柱面 0、读写头/盘面 0,扇区 1,下一个扇区编址为柱面 0,磁头/盘面 0,扇区 2,依此类推。

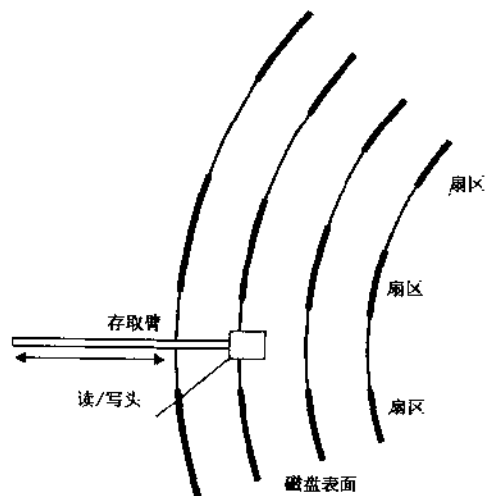


图 16-1 磁盘表面和读写头

2. 相对分区法,扇区是相对于磁盘的起始位置来编号的,所以磁盘上在柱面 0、轨道 0 上的第 1 个扇区被编址为相对扇区 0,下一个为相对扇区 1,直到磁盘的最后一个扇区。

不同的磁盘操作可能使用不同的方法,这取决于执行何种访问方式。

16.2.3 磁盘控制器

位于处理器和磁盘驱动器之间的磁盘控制器处理它们之间所有的通信。控制器接收来自处理器的数据并转换成驱动器可用的格式。例如,处理器会发出一个请求,数据来自一个指

定的柱面一头一扇区。控制器的任务是提供一个适当的命令去移动取数臂到所要求的柱面, 并传输数据。并让数据到计算机, 计算机接收数据。

当控制器工作时, 处理器对其他任务而言是空闲的。在这种方式下, 控制器一次只处理一位。然而, 控制器也可以执行更快的 I/O, 这时数据完全不通过处理器, 直接和存储器传输数据, 这种传输大量数据的方式称为直接存储器存取(DMA)。为了这个目的, 处理器给控制器提供读或写的命令、存储器中 I/O 缓冲区的地址、要传输数据的扇区号、柱面号、磁头以及起始扇区。用这种方法, 处理器必须等待直到 DMA 完成, 因为同一时间只能有一个部件使用存储器通道。

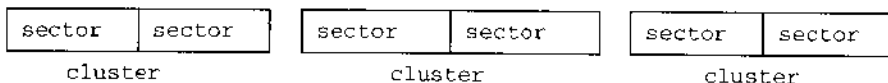
有两个因素影响数据传输速率, 即磁盘驱动器传递数据到计算机的速度: 存取时间和旋转速率。

1. 存取(或寻找)时间与磁头到达所要求的柱面/磁道的移动速度有关。对于顺序处理, 读写头最多移动一个柱面; 对于随机处理, 读写头可以移动多个柱面。

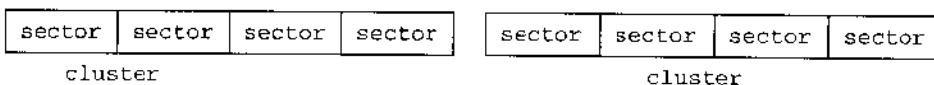
2. 旋转速率决定了所要求的扇区到达磁头的时间以及从扇区传输数据到计算机存储器的时间。当然这个操作的平均时间有一半是在旋转, 这被认为是等待时间。对于旋转速率为 6000 rpm 的情况, 每秒的速度是 $6000/60=100$ 转数。旋转一周需要 $1/100$ 秒, 相当于 10 毫秒, 因此等待时间为 5 毫秒。

16.2.4 簇

簇是一组扇区, 系统将簇当作存储空间单位。一个簇的大小总是 2 的乘方, 比如 1, 2, 4 或 8 个扇区。对于支持每簇一个扇区的软盘驱动器, 扇区和簇是一样的。每簇两个扇区的磁盘组织如下:



每簇 4 个扇区的磁盘组织如下:



硬盘驱动器可以被分成几个分区, 每个分区通过驱动器号来识别, 第一个分区是 C。硬盘上的 FAT(文件分配表)有两种类型, FAT16 和 FAT32, 根据以下规则来确定一个硬盘簇的大小:

FAT16			FAT32		
分 区	簇大小	扇区号	分 区	簇大小	扇区号
少于 128MB	2KB	4	250MB ~ 8GB	4KB	8

一个 400 字节的文件(足够小,可以放在一个扇区中)存储在每簇 64 扇区的磁盘,占用了 $64 \times 512 = 32\,768$ 字节的存储量,尽管实际上只有一个扇区含有数据。未使用的(和目前不可使用的)磁盘空间被认为是分散的。

对于每个文件,FAT 按升序存储它的簇,尽管文件可能会被分成碎片保存,例如,分别在簇 8,9,10,14,17 和 18 中保存。一个簇也可以从一个磁道交送到另一个磁道。

磁盘容量。这里是两种 3.5" 磁盘的存储容量:

存储容量	磁道数/面(柱面)	扇区数/道	字节数/扇区	双面总字节	扇区/簇
3.5" 720KB	80	9	512	737,280	2
3.5" 1.44MB	80	18	512	1,474,560	1

对于硬盘,容量通常随着驱动器和分区变化。确定柱面、每个磁道的扇区数或读写头等一些很有用的操作,包括 INT 21H 功能 1FH 和 440DH 的子功能 60H,这些都会在第 18 章中介绍。

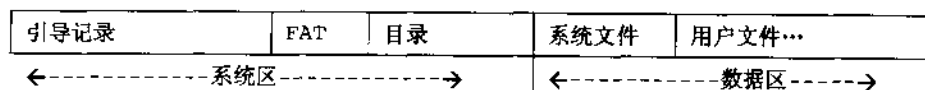
16.3 磁盘系统区和数据区

保留某些扇区的目的是为了提供磁盘上有关文件的信息。软盘和硬盘的组织因它们的容量而不同。硬盘和一些软盘格式化为自引导的——也就是说,它们在接通电源或者用户按下了 Ctrl+Alt+Del 键时可以自行启动。磁盘的组织一般由系统区以及紧接着的数据区组成,磁盘其余的空间都是数据区。

16.3.1 系统区

系统区是磁盘上的第一个区,从最外面的磁道开始,即 0 面,0 磁道,1 扇区。系统区存放系统存储和维护的信息,例如每个文件存储在磁盘上的起始位置等。系统区的 3 个组成部分是(1)引导记录,(2)文件分配表(FAT),(3)目录。

系统区和数据区的组织如下:



下面的表给出了 3.5" 磁盘设备的组织结构,说明了引导记录、FAT 和目录的开始和结束的扇区号。扇区由相对扇区号识别,相对扇区 0 就是柱面 0,轨道 0,扇区 1,也就是设备上的第一个扇区(早先在“柱面”那一节解释过):

设备	引导记录	FAT	目录	扇区数/簇
3.5" 720KB	0	1-6	7-13	2
3.5" 1.44MB	0	1-18	19-32	1

对于硬盘来说,引导记录和 FAT 的位置通常和软盘的一样,而 FAT 的大小和目录的位置是随着设备不同而变化的。

一张格式化的软盘包含以下信息(按照起始的物理和相对扇区):

文件	720K (9 扇区/磁道)				1.44MB (18 扇区/磁道)			
	柱面	盘面	扇区	相对扇区	柱面	盘面	扇区	相对扇区
引导记录	0	0	1	0	0	0	1	0
FAT1	0	0	2	1	0	0	2	1
FAT2	0	0	4	4	0	0	11	10
目录	0	0	8	7	0	1	2	19
数据区	0	1	6	1	0	1	16	33

在 720K 软盘上的数据文件开始于柱面 0、盘面 1、扇区 6—9。系统存储下一个记录就在柱面 1、盘面 0，然后在柱面 1、盘面 1，接着在柱面 2、盘面 0，以此类推。在前移到下一个柱面之前，在相对的两个盘面上填写数据的特性减少了磁盘读写头的移动，这种方法既用于软盘也用于硬盘。

16.3.2 数据区

在可引导磁盘或软盘的数据区以 IO.SYS 和 MSDOS.SYS 两个系统文件开始。当用 FORMAT/S 格式化磁盘时，DOS 就把它的系统文件复制到数据区的第一个扇区。用户文件紧跟着系统文件，或者在没有系统文件的情况下从数据区的起始位置开始。

下一节解释引导记录、目录和 FAT。

16.4 引导记录

引导记录包含了把系统文件(如果存在)从磁盘载入(或“导入”)存储器的指令。所有格式化的磁盘都包含一个引导记录，即使系统文件没有存储在磁盘上。引导记录包含了下面以偏移地址为序的信息：

00H	跳转到引导记录中偏移地址为 3EH 的自引导程序
03H	创建引导程序时的名字或 DOS 版本号
0BH	每扇区的字节数，通常为 200H(512)
0DH	每簇的扇区数(1、2、4 或 8)
0EH	保留的扇区数
10H	FAT 的备份数(1 或 2)
11H	根目录的入口数
13H	容量小于 32MB 的磁盘扇区数
15H	媒体描述符字节(和 FAT 的第一个字节相同，以后会解释)
16H	FAT 的扇区数
18H	每个磁道的扇区数
1AH	读写头数(盘面或盘表面)

1CH	隐藏扇区数
1EH	系统保留
20H	扇区总数, 如果容量大于 32MB
24H	物理驱动器数(软盘: A=0; 硬盘: 80H=驱动器 C, 等等)
25H	系统保留
26H	扩展引导扇区标记(包含 29H)
27H	卷 ID
2BH	卷标
36H	系统保留
3EH-1FFH	自引导程序的超始地点

16.5 目 录

磁盘上的所有文件从一个簇的边界开始, 也就是这个簇的第一个扇区。对于每一个文件, 系统建立了一个 32 字节(20H)的目录项来描述文件的名称、创建日期、文件大小以及文件起始簇的位置。目录项的格式如下:

字节	目的
00H-07H	文件名, 在创建文件的程序中定义。第一个字节也可以表明文件的状态: 00H 文件从未使用过 05H 文件名的第一个字符实际上是 E5H 2EH 文件是个子目录 E5H 文件已被删除
08H-0AH	文件扩展名, 例如 EXE 或 ASM
0BH	文件属性, 定义文件类型(注意, 一个文件可以有多个属性): 00H 正常属性文件 01H 只能被读出的文件(只读文件) 02H 隐藏文件, 目录搜索不显示 04H 系统文件, 目录搜索不显示 08H 卷标(如果这是个卷标记录, 标签本身也在文件名和扩展名的字段中) 10H 子目录 20H 存档文件, 表明从最后一次更新后文件是否修改过。 (例如, 代码 07H 说明系统文件(04H)是只读的(01H)和隐藏的(02H)。)
0CH-15H	系统保留
16H-17H	时间, 文件创建或最后一次修改的时间: 以二进制格式存储 16

- 位, 如 hhhhhmmmmmmsssss。
- 18H-19H 日期, 文件创建或最后一次修改的日期, 以二进制格式存储 16 位, 如 yyyyyymmmmdddd。年为 000-119(1980 作为起始点), 月为 01-12, 日为 01-31。
- 1AH-1BH 文件起始簇。其编号与目录的最后两个扇区有关。如果没有系统文件, 第一个数据文件从相对簇 002 开始。实际的盘面、磁道和簇取决于磁盘容量。一个为零的项说明文件没有给它空间分配。
- 1CH-1FH 文件大小, 以字节计数。写文件后, 系统计算并保存文件的大小在这个字段中。

对于在目录中超过一个字节的数字字段, 数据按字节以反序存储。

16.6 文件分配表

FAT 的目的是为文件分配磁盘空间。FAT 包含有磁盘上每个簇的入口项。当创建一个新文件或修改一个已存在的文件时, 系统根据磁盘上文件的位置修改相关的 FAT 入口项。FAT 开始于扇区 2, 紧接着是引导记录。对一个簇由 4 个扇区组成的磁盘, 相同编号的 FAT 入口可以访问 4 次, 这和一簇一个扇区的磁盘的数据量一样。因此, 使用多扇区的簇减少了 FAT 的入口数, 并且能使系统访问一个更大的磁盘存储空间。

最初的设计者提供了 FAT 的两个拷贝(FAT1 和 FAT2), 大概是因为若 FAT1 被破坏可以使用 FAT2。但是, 虽然仍保留着 FAT2, 但它却从未起作用。前面“磁盘系统区和数据区”一节在 FAT 存储要求中包括了 FAT1 和 FAT2。本书中其他所有讨论都涉及到 FAT1。

16.6.1 FAT 的第一个入口项

FAT 的第一个字节为媒体描述符, 它指出设备类型(类似于引导记录中的字节 15H)如下:

F0H 3.5", 双面, 18 扇区/磁道(1.44MB)和 3.5", 双面, 36 扇区/磁道(2.88MB)

F8H 硬盘(包括 RAM 磁盘)

F9H 3.5", 双面, 9 扇区/磁道(720KB)和 5.25", 双面, 15 扇区/磁道(1.2MB)

注意, F0H 和 F9H 分别指定了两种不同的磁盘格式。

16.6.2 FAT 的第二个入口项

对于支持 12 位 FAT 入口的软盘 FAT, 第二个入口项含有 FFFFH; 对于支持 16 位 FAT 入口的硬盘, 第二个入口项含有 FFFFFFFFH。头两个入口项如下:

1.44MB diskette

F0	FF	FF
----	----	----	----	----	----	----	----	----	----

Hard disk

F8	FF	FF	FF
----	----	----	----	----	----	----	----	----	----

正如已经描述过的,磁盘的第一个字段是引导记录,紧跟的是 FAT 和目录,接着就是数据区。入口项图示如下:

簇 0	簇 1	簇 2	簇 3	...	簇 n
<-----目录区----->		<-----数据区----->			

你可能希望数据区就在簇的起始点,然而,头两个簇(0 和 1)指向了目录,这样存储数据文件的数据区就起始于簇 2。

16.6.3 FAT 的指针入口项

在头两个 FAT 入口项之后的是指针入口项,它涉及到数据区的每个簇。目录(在 1AH-1BH)包含文件第一个簇的位置,并且 FAT 包含了每个后继簇的指针入口链。

软盘的入口项长度是 3 位十六进制数(1 1/2 字节,或 12 位),对于硬盘是 4 位十六进制数(两字节,或 16 位)。Windows 也提供了 32 位的 FAT 入口项。每个 FAT 指针入口项根据以下格式指出特定簇的用法:

12 位	16 位	说 明
000	0000	基准簇目前未使用
nnn	nnnn	与文件下一簇相关的簇号
FF0-FF6	FFF0-FFF6	保留簇
FF7	FFF7	不能用(坏磁道)
FFF	FFFF	文件最后一簇

对于 1.44MB 软盘(12 位 FAT)的前两个入口项包含 F0F 和 FFn, 分别表示簇 0 和簇 1:

FAT 入口项:	F0F	FFF	-	-	-	-	-	-	-
相对簇:	0	1	2	3	4	5	6	...	End

开始的 F0 表示双面、9 扇区(1.44MB)的软盘,随后是 FFFFH。术语“相对簇”的含义是,例如, FAT 的第三个指针入口项指向相对簇 2,第四个入口项指向相对簇 3,依此类推。在某种意义上,前两个 FAT 入口项(相对簇 0 和 1)指向目录的最后两个簇,它们已经被分配在簇的开始,目录指出文件的大小和起始簇。

目录包含每个文件的起始簇号以及指向下一簇位置的 FAT 指针入口链,正因为有指针链,才使文件得以连续。含有(F)FFFH 的指针入口项表示为文件的最后一簇。

FAT 入口项举例。以下的例子应该能帮助阐明 FAT 的结构。假设磁盘只包含一个文件,名字叫 TEMPSTAT.FIL,它全部存储在簇 2、3 和 4。这个文件的目录入口项包含文件名 TEMPSTAT,扩展名 FIL,00H 表明是个正常属性文件,创建日期,0002H 是文件第一个相对簇的位置,以及表示文件字节大小的入口项。12 位的 FAT 入口项表示如下,除了有一对字节应当是反序的:

FAT 入口项:	F0F	FFF	003	004	FFF
相对簇:	0	1	2	3	4	5	6	...	end

一个程序把 TEMPSTAT.FIL 连续地从磁盘读到存储器, 系统按以下步骤执行:

- 对于第一簇, 搜索磁盘目录寻找文件名为 TEMPSTAT、扩展名为 FIL 的文件, 从目录获取文件的第一个相对簇(2)的位置, 并且传递其内容(来自扇区的数据)到主存储器中的程序。
- 对于下一簇, 访问用相对簇 2 表示的 FAT 指针入口项。从上图可知, 这个入口项包含 003, 这说明文件在相对簇 3 上继续。系统传递这一簇的内容到程序。
- 对于最后一簇, 访问用相对簇 3 表示的 FAT 指针入口项。这个入口项包含 004, 说明文件在相对簇 4 上继续。系统传递这个簇上的内容到程序。
- 相对簇 4 的 FAT 入口项包含 FFFH, 指出没有簇分配给文件了。系统现在从簇 2、3 和 4 上传递了所有文件的数据。

我们已经了解了 FAT 入口项工作的原理, 现在再看它们如何以反序字节来工作的, 这里需要一点灵活性。

16.6.4 处理 12 位 FAT 入口项

下面是与刚才介绍过的 TEMPSTAT.FIL 相同的一个例子, 不过现在使用的是反序字节的指针入口项。这个文件的 12 位 FAT 如下:

FAT 入口项:	F0F	FFF	034	000	FF0	Fxx	...
相对簇:	0	1	2	3	4	5	

但是现在需要辨识这些入口项, 因为它们是根据相对字节而不是簇来表示的:

FAT 入口项:	F0	FF	FF	03	40	00	FF	0F	...
相对字节:	0	1	2	3	4	5	6	7	

下面是访问簇的步骤:

- 为了处理第一个 FAT 入口项, 2(文件的第一个簇记录在目录中)乘 1.5(FAT 入口项的长度)得到 3。(程序中乘 3 并右移一位。)存取 FAT 中字节 3 和 4 中的字。这两个字节包含 03 40, 反序时是 4003。因为簇 2 是个偶数, 使用后 3 位数字, 所以 003 是文件的第二簇。
- 对于第三簇, 簇号 3 乘 1.5 得 4。存取 FAT 中的字节 4 和 5。这些字节包含 40 00, 当反序时为 0040。因为簇 3 为一个奇数, 使用前 3 位数字, 所以 004 是文件的第三簇。
- 对于第四簇, 簇 4 乘 1.5 得 6。存取 FAT 中的字节 6 和 7。这些字节包含 FF 0F, 当反序时, 是 0FFF。因为簇 4 是一个偶数, 使用后 3 位数字 FFF, 它说明这是最后的入口项。

16.6.5 处理 16 位的 FAT 入口项

先前已经提到, 硬盘 FAT 的第一个字节包含有媒体描述符。这个字节之后是 FFFFFFFH, 正如下表中所表示的。FAT 指针入口项是 16 位字长, 并且开始于字 3 和字 4, 它表示为簇 2。目录入口项提供了文件的起始簇。虽然每个入口项的字节是反序排列的, 从每个 FAT 入口项确定的簇号是简单的。

作为一个 16 位 FAT 入口项的例子,假设一个指定硬盘上惟一的一个文件占用了 4 个簇(每簇 4 个扇区,总共 16 个扇区)。根据目录知道,文件从簇 2 开始。每个 FAT 指针入口项是一个完整的字,所以反序的字节只包括一个入口项。这里是指针入口项以反序字节排列的 FAT:

FAT 入口项:	F8FF	FFFF	0300	0400	0500	FFFF	...
相对簇:	0	1	2	3	4	5	

相对簇 2 的 FAT 入口项为 0300,反序后为下一个簇 0003。相对簇 3 的 FAT 入口项为 0400,反序后是下一个簇 0004。用保留的入口链这种方式继续直到 5 号簇。指针入口项 FFFFH 指向文件尾。32 位入口项的 FAT 使用反序字节排列的双字。

如果程序要确定安装的磁盘类型,它可以直接检查引导扇区的媒体描述符或使用 INT 21H 的功能 36H。

16.6.6 练习:检测 FAT

让我们使用 DEBUG 来检测磁盘 FAT。这个练习需要一张容量为 1.44MB 的格式化了空白 3.5"软盘,而且上面没有拷贝系统文件。复制两个文件到磁盘上,第一个文件要大于 512 字节、小于 1024 字节,以占用 2 个扇区,建议用 A04ASM1.ASM。第二个文件要大于 1536 字节且小于 2048 字节,以占用 4 个扇区,建议用 A09DRVID.ASM。由于文件从偏移地址 100H 开始存储,因此可以用下列方法定位记录:

1. 引导记录在起始地址 100H。
2. FAT 在引导记录扇区之后: $100H + 200H(1 \text{ 扇区}, 512 \text{ 或 } 200H \text{ 字节}) = 300H$ 。
3. 目录在 FAT 之后: $300H + [18(12H) \text{ 扇区} \times 200H] = 2700H$

首先把软盘插入驱动器 A。装入 DEBUG 并键入 L(load)命令(附录 E 有更完全的解释):

L 100 0 0 30 (对于驱动器 B, 使用 L 100 1 0 30)

L 命令的入口项是:

- 100H 是 DEBUG 段的起始偏移地址,数据将从这里读入。
- 第一个 0 意味着使用驱动器 A。
- 第二个 0 意味着从相对扇区 0 开始读数据。
- 30 表示从 30H(48)扇区读取数据。

键入命令 D 100 开始显示。现在可以检测磁盘的引导记录、目录和 FAT。

• 引导记录。引导记录的一些字段为:

- 段偏移地址 103H 中是建立 FAT 时的 DOS 版本。
- 偏移地址 10BH 中是每扇区的字节数(0002H 反序字节为 0200H, 或 512 字节)。
- 偏移地址 115H 是该磁盘的媒体描述符 F0H。再检验其他字段。

• 目录。检测目录,键入命令 D 2700:

- 在偏移地址 2700H 包含第一个文件的文件名, A04ASM1.ASM。
- 在偏移地址 271AH 中给出这个文件的起始簇号(0200 或 0002)。
- 在偏移地址 271CH~271FH 中给出文件的大小(27030000 或 327H 字节)。
- 偏移地址 2720H 开始是第二个文件 A10DRVID.ASM 的入口项。注意, 273AH 显示

了它的起始簇为 0400 或 0004。文件大小为 673H 字节。

3. FAT。检测 FAT，键入命令 D 300，将显示：

FAT 入口:	F0	FF	FF	03	F0	FF	05	60	00	07	F0	FF	...
相对字节:	0	1	2	3	4	5	6	7	8	9	10	11	

- F0 是媒体描述符。
- 字节 1 和字节 2 中的 FF FF 是第二个字段的内容。

从字节 3 开始的指针入口项可以分析如下：

- 对于第一个文件，2(根据目录得知它的第一簇)乘 1.5 得到相应字节 3。访问 FAT 的偏移字节 3 和 4，它包含 03 F0，取反序字节得到 F003。因为簇 2 是个偶数，使用最后 3 位数字 003，这是系列中的下一簇。簇 3×1.5 是 4，相应字节 4 和 5 包含 F0 FF，取反序字节为 FFF0。因为簇 3 是个奇数，使用头 3 个数字 FFF，这表示文件结束。这样就知道文件驻留在簇 2 和簇 3 中了。

- 对于第二个文件，第一簇是 4：

簇	FAT 偏移	FAT 值	簇	位数	下一簇
4×1.5	6&7	60 05	偶	后 3 位	005
5×1.5	7&8	00 60	奇	前 3 位	006
6×1.5	9&10	F0 07	偶	后 3 位	007
7×1.5	10&11	FF F0	奇	前 3 位	FFF

FFF 表示数据结束。

INT 21H 给程序提供了一些支持服务例程以访问目录和 FAT 的信息，包括功能 47H(获取当前目录)，有关说明在第 18 章。

16.7 处理磁盘文件

数据以文件的形式存储在磁盘上，就像你已经存储的程序一样。虽然没有限制保存在文件中的数据的数据的类型，但是一个典型的用户文件由客户记录、库存供应或姓名地址表组成。每个记录包含了有关详细的客户信息或项目清单信息。在文件里，所有记录通常(但不是必须)有相同的长度和格式。一个记录包含一个或多个提供有关记录信息的字段。例如，对一个客户文件的记录，可以包含比如客户号，姓名，地址和应付款这些字段。记录按客户号升序排列。

处理硬盘上的文件和软盘上的文件大致相同，对于两者，你必须提供一个路径名来存取子目录中的文件。

有许多的特殊中断服务支持磁盘输入/输出。一个程序写(或建立)一个文件首先使系统在目录里生成一个入口项。当所有文件的记录写完，程序关闭文件，这样系统就可以完成文件大小的目录入口项。

程序读一个文件首先要打开文件以确保它是存在的。一旦程序读完全部记录，应当关闭

这个文件，使它对其他程序也是可用的。由于目录的设计，你可以顺序处理(连续的一个记录接着一个记录)磁盘文件的记录，也可以随机处理(在文件范围内按要求检索记录)磁盘文件的记录。

最高级磁盘处理是通过 INT 21H，借助于目录和记录的“分块”、“解块”来支持磁盘处理。这种方法在连接到 BIOS 之前执行了一些预备处理。第 17 章涉及到 INT 21H 的写和读磁盘文件操作的用法。第 18 章讨论了各种支持目录和磁盘文件的操作。

最低级磁盘处理是通过 BIOS 中断 13H，其包括了磁道和扇区号的直接寻址，这将在第 19 章中讲到。

16.8 重 点

- 软盘或硬盘的每面都包含了许多同心的磁道，磁道号从 00 开始。每个磁道格式化为 512 字节的扇区，扇区号从 1 开始。

- 柱面是每个盘面上所有相同编号磁道的集合。
- 程序可以通过柱面-读写头或相对扇区号来定位一个扇区。
- 簇是一组扇区，系统把簇看作是存储空间单位。簇的大小总是 2 的乘方，比如 1、2、4 或 8 扇区。

- 无论文件大小如何，所有文件都起始于一个簇的边界并且至少需要一个簇。
- 引导记录包含把系统文件从磁盘载入(或“导入”)到内存的指令。
- 目录包含磁盘上每个文件的入口项，并且指出文件名、扩展名、文件属性、时间、日期、起始扇区和文件大小。

- 文件分配表(FAT)用来为文件分配磁盘空间。FAT 起始于扇区 2，紧跟在引导记录之后，并且为目录中的每个文件包含了每个簇的一个入口项。

16.9 习 题

16-1. 一个标准扇区有多少字节？

16-2. 什么是柱面？

16-3. 磁盘控制器的用途是什么？

16-4. (a)什么是簇？(b)簇的用途是什么？(c)每簇大小分别为 1、2、4 或 8 扇区的磁盘空间(以字节为单位)是多少？

16-5. 用柱面数、每磁道的扇区数和每扇区的字节数说明怎样计算以下软盘的容量：

(a) 3.5", 720K 软盘 (b) 3.5", 1.44MB 软盘。

16-6. 磁盘系统区的 3 部分是什么？

16-7. (a)引导记录的目的是什么？(b)引导记录位于什么地方？(c)如何利用它来确定每个磁道的扇区数？

16-8. 目录如何指示一个被删除的文件？

16-9. 目录中对于 (a) 普通文件, (b) 只读文件, (c) 系统文件, (d) 卷标, (e) 只读隐藏文件的指示是什么?

16-10. 当使用 `FORMAT/S` 格式化磁盘时, 对软盘或硬盘的附加作用是什么?

16-11. 假定有一个大小为 2 259(十进制)字节的文件。(a) 系统把文件大小存储在什么地方? (b) 用十六进制形式表示文件大小是多少? 当系统存储文件时, 显示这个值。(c) 这个文件需要多少扇区?

16-12. `FAT` 在什么地方, 以及如何指出它所驻留的设备为 (a) 硬盘, (b) 3.5", 720K 软磁盘, (c) 3.5", 1.44MB 软磁盘?

16-13. `FAT` 如何指示 12 位入口项和 16 位入口项?

磁盘存储 II: 写文件和读文件

目的: 介绍文件代号的使用以及顺序和随机读写磁盘文件的操作。

17.1 引言

本章介绍了写磁盘文件和读磁盘文件的中断服务, 第 18 章介绍支持处理磁盘驱动器、目录和文件的各种服务请求。其中有许多磁盘操作包括了用 ASCIIZ 串来识别驱动器、路径和文件名; 连续存取文件的文件代号; 专门用来识别错误的返回码。

本章虽然不要求新的汇编语言指令, 但是介绍了下面一些处理磁盘文件的 INT 21H 服务例程:

3CH	建文件	3FH	读记录
3DH	打开文件	40H	写记录
3EH	关闭文件	42H	移动文件指针

这里要注意, 术语“簇”表示一组扇区, 它含有一个或多个数据区, 这取决于设备。

17.2 ASCIIZ 串

当要用到许多磁盘处理的扩展服务例程时, 首先要给系统提供一个 ASCIIZ 串的地址, ASCIIZ 串包含文件规范: 磁盘驱动器的位置、目录路径和文件名(都为可选项和内部省略符), 紧接着是一个十六进制的 0 字节。这个串的最大长度是 128 字节。

下例定义了一个驱动器和文件名:

```
PATHNAM1 DB 'C:\A17RANRD.ASM', 00H
```

这个例子定义了驱动器、子目录和文件名:

```
PATHNAM2 DB 'C:\UTILITY\A17RANRD.EXE', 00H
```

串中的后斜线也可以是前斜线, 它起到路径分隔符的作用。十六进制的 0 字节用来结束串。对于一个请求 ASCIIZ 串的中断服务, 要把它偏移地址装入 DX 寄存器, 例如: LEA DX,

PATHNAM1。

17.3 文件代号

正如在第 8 章讨论的,可以对某一个标准设备直接使用文件代号:00=输入设备,01=输出设备,02=错误输出设备,03=辅助设备,04=打印机。其他 I/O 服务例程包括使用文件代号存取文件的操作,对这些操作,必须从系统请求文件代号。对一个磁盘文件必须先要打开,这和对键盘或屏幕传输数据不一样,系统必须通过目录和 FAT 中的入口来寻址磁盘文件,还要修改这些入口。与程序有关的每个文件在执行时必须分配到一个属于它自己的唯一的文件代号。

当为了输入打开一个文件或为了输出建立一个文件时,系统就传递出一个文件代号。这些操作包括使用 ASCIIZ 串和 INT 21H 的功能 3CH 或 3DH。返回在 AX 中的文件代号是有惟一性的一个字长的数码,你要把这个文件代号保存在一个字数据项中,在以后请求存取文件时就使用这个文件代号。一般情况下,第一个返回的文件代号是 05,第二个是 06,以此类推。

PSP 含有一个默认的文件代列表,它提供了近 20 个文件代号(这是对打开文件的一种象征性的限制),可以用 INT 21H 的功能 67H 来扩大这个限度,这在第 23 章中解释。

17.4 错误返回码

磁盘的文件代号操作通过进位标志和 AX 寄存器传递出一个完成的状态。若操作成功,则清除进位标志为 0,并执行其他相关的功能。如操作不成功,则设置进位标志为 1,并根据操作在 AX 中返回一个错误码。图 17-1 列出了错误码 01—36,其他的代码与网络有关。如果这些错误码还不够,INT 59H 提供了有关错误的附加的信息(参见第 18 章)。

01	非法功能号	20	未知单元
02	文件未发现	21	驱动器未准备好
03	路径未找到	22	未知命令
04	打开的文件太多	23	CRC 数据错
05	拒绝存取	24	请求指令长度错
06	非法文件代号	25	搜索错
07	内存控制块被破坏	26	未知的介质类型
08	内存不足	27	未发现扇区
09	无效存储块地址	28	打印机纸出界
10	无效环境	29	写故障
11	非法格式	30	读故障
12	非法存取代码	31	一般性失败
13	无效数据	32	共享违例
15	指定的设备无效	33	锁违例
16	试图删除目录	34	非法磁盘更换
17	设备不一致	35	FCB 无效
18	已没有文件	36	共享缓冲区溢出
19	磁盘写保护		

图 17-1 主要的磁盘错误返回码

17.5 文件指针

系统为正在处理的程序的每个文件保留了一个单独的文件指针。建立和打开文件操作把文件指针的值初始化为 0，即文件的起始位置。文件指针对文件内当前的偏移地址不断地进行计算。

每个读/写操作都会使系统通过传输的字节数对文件指针增量，然后文件指针就指向要存储的下一个记录的位置。文件指针对顺序和随机处理都很方便。对记录的随机处理，程序可以使用 INT 21H 的功能 42H（在后面一节中要讲到）把文件指针设置到文件的任何位置。

下面几节包括了建立磁盘文件、写磁盘文件和关闭磁盘文件的中断请求。

17.6 建立磁盘文件

写磁盘的过程如下：

1. 用 ASCII 串从系统得到文件代号；
2. 用 INT 21H 的功能 3CH 建立文件的目录入口；
3. 用 INT 21H 的功能 40H 写文件记录；
4. 最后，用 INT 21H 的功能 3EH 关闭文件。

17.6.1 INT 21H 的功能 3CH：建立文件

建立一个新文件或用相同的名字重写一个旧文件时，首先要使用 INT 21H 的功能 3CH。把请求的文件属性装入 AX（第 16 章中已介绍过），ASCII 串的地址装入 DX（新文件在磁盘上的位置）。下例在驱动器 C 建立一个属性为 0 的正常文件：

```

PATHNAM1  DB  'C:\ACCOUNTS.FIL', 00H
FILHAND1  DW  ?                      ; 文件代号
...
MOV        AH, 3CH                    ; 请求建立文件
MOV        CH, 00                     ; 正常属性
LEA        DX, PATHNAM1               ; ASCII 串
INT        21H                        ; 调用中断服务例程
JC         error                       ; 出错时的特定动作
MOV        FILHAND1, AX                ; 保存文件代号到一个字中

```

对一个有效的操作，系统用给定的属性建立一个目录入口，清除进位标志，并在 AX 中设置文件代号。以后存取文件都使用这个文件代号。打开指定文件的同时，把文件指针设置为 0（文件的开始），这时就可以写文件了。如果文件在指定的路径上已经存在，该操作将文件长度置为 0，使新文件对旧文件进行重写。

对于错误的情况，操作设置进位标志，并在 AX 中返回错误码：03、04 或 05（见图 17-1）。

错误码 05 说明目录满了或者涉及到的文件名具有只读的属性, 首先应当检验进位标志。例如, 建一个文件时可能给 AX 传送的文件代号是 05, 这很容易与拒绝存取的错误码 05 相混淆。有关建立文件的服务是 INT 21H 的功能 5AH 和 5BH, 这些内容将在第 18 章介绍。

17.6.2 INT 21H 的功能 40H: 写记录

INT 21H 的功能 40H 用于在磁盘上写记录。在 BX 中装入保存的文件代号。CX 中是要写入的字节数, DX 中是输出区的地址。下例使用文件代号完成从建文件操作到写入 DSKAREA 中的 256 字节的记录:

```

FILHAND1  DW  ?           ; 文件代号
DSKAREA   DB  256 DUP ( ' ' ) ; 输出区
...
MOV  AH, 40H           ; 请求写记录
MOV  BX, FILHAND1      ; 文件代号
MOV  CX, 256           ; 记录长度
LEA  DX, DSKAREA       ; 输出区地址
INT  21H               ; 调用中断服务例程
JC   error1            ; 出错时的特定动作
CMP  AX, 256           ; 全部字节写完?
JNE  error2            ; 未完, 则出错

```

一次有效的操作将把记录写到磁盘上、对文件指针增量、清除进位标志、并设置 AX 为实际写入的字节数。一个已写满的磁盘可能会使实际写入的字节数和要求写入的字节数不同, 但是因为系统没有把这种情况作为一个错误报告, 因此程序必须测试 AX 中的返回值。非法操作将设置进位标志为 1, 并返回给 AX 错误码 05 (拒绝存取) 或 06 (非法文件代号)。

17.6.3 INT 21H 的功能 3EH: 关闭文件

完成写磁盘文件后, 程序必须将这个文件关闭。在 BX 装入文件代号, 并调用 INT 21H 的功能 3EH:

```

MOV  AH, 3EH           ; 请求关文件
MOV  BX, FILHAND1      ; 文件代号
INT  21H               ; 调用中断服务例程
JC   error             ; 测试错误

```

成功的关闭操作将仍在内存缓冲区中的剩余记录写入磁盘, 并用日期和文件大小修改 FAT 和目录。不成功的操作将设置进位标志, AX 中的返回码只可能是错误码 06 (非法文件代号)。

17.6.4 程序: 建立一个磁盘文件

图 17-2 的程序建立一个由用户键入姓名的文件。它的主要过程如下:

- A10MAIN 调用 B10CREATE, C10PROC, 如果输入结束调用 E10CLOSE。
- B10CREATE 用 INT 21H 的功能 3CH 建立文件, 并将文件代号保存在一个数据项 FILEHAND 中。
- C10PROC 从键盘接收输入, 并把姓名之后剩余的输入区清 0。
- D10WRITE 利用 INT 21H 的功能 40H 写记录。
- E10CLOSE 利用 INT 21H 的功能 3EH 在处理结束时关闭文件以建立相应的目录入口。
- F10DISPLY 在屏幕上显示数据。

```

TITLE      A17CRFIL (EXE) 建立姓名的磁盘文件
.MODEL     SMALL
.STACK     64
.DATA
NAMEPAR    LABEL    BYTE           ; 参数表:
MAXLEN     DB        30           ; 最大长度
NAMELEN    DB        ?           ; 实际长度
NAMEREC    DB        30 DUP(' '), 0DH ; 输入姓名
ERRCODE    DB        00           ; 错误标识
FILEHAND   DW        ?           ; 文件代号
PATHNAME   DB        'C:\NAMEFILE.DAT', 0
PROMPT     DB        'Name? '
OPENMSG    DB        '*** Open error  ***'
WRITEMSG   DB        '*** Write error ***'
ROW        DB        0           ; 屏幕行
.386 ; -----
.CODE
A10MAIN    PROC    FAR
MOV        AX,@data           ; 初始化
MOV        DS,AX              ; 段寄存器
MOV        ES,AX              ; 寄存器
MOV        AX,0003H           ; 设置显示方式
INT        10H                ; 并清屏
CALL       B10CREATE           ; 建文件
CMP        ERRCODE,00          ; 建文件错?
JNZ        A90                 ; 是, 退出
A20:       CALL      C10PROC     ; 取文件代号
CMP        NAMELEN,00          ; 输入结束?
JNE        A20                 ; 否, 继续
CALL       E10CLOSE            ; 是, 关文件
A90:       MOV        AX,4C00H   ; 结束处理
INT        21H
A10MAIN    ENDP

; ----- 建立磁盘文件, 测试是否有效: -----
B10CREATE  PROC
MOV        AH,3CH              ; 使用 AX, BP, CX, DX
MOV        CX,00               ; 请求建文件
MOV        DX,PATHNAME         ; 正常属性文件
INT        21H
JC         B20                  ; 出错?
MOV        FILEHAND,AX         ; 否, 保存文件代号
JMP        B90
B20:       LEA        BP,OPENMSG ; 是, 出错
MOV        CX,19               ; 信息长度
CALL       F10DISPLY
MOV        ERRCODE,01          ; 设置错误代码
B90:       RET
B10CREATE  ENDP

; ----- 从键盘接收姓名: -----
C10PROC    PROC
MOV        CX,06               ; 使用 AX, BP, CX, DI, DX
LEA        BP,PROMPT           ; 提示信息长度
CALL       F10DISPLY           ; 显示提示信息

```

图 17-2 建立一个磁盘文件

```

MOV     AH,0AH           ; 请求输入
LEA     DX,NAMEPAR       ; 接收姓名
INT     21H
CMP     NAMELEN,00       ; 有姓名输入?
JE      C90              ; 否,退出
MOV     AL,20H           ; 存储空格
MOVZX   CX,NAMELEN       ; 长度
LEA     DI,NAMEREC
ADD     DI,CX            ; 地址+长度
NEG     CX               ; 计算
MOVZX   DX,MAXLEN        ; 剩余的
ADD     CX,DX            ; 长度
REP     STOSB            ; 设置空格
CALL    D10WRITE         ; 写磁盘记录
C90:    RET
C10PROC ENDP
;
; ----- 写磁盘记录, 测试是否有效: -----
;
D10WRITE PROC NEAR       ; 使用 AH, BP, BX, CX, DX
MOV     AH,40H           ; 请求写记录
MOV     BX,FILEHAND
MOVZX   CX,MAXLEN        ; 30 字节姓名为
ADD     CX,2             ; 加上 2 字节为回车/换行
LEA     DX,NAMEREC
INT     21H
JNC     D20              ; 写文件有效?
LEA     BP,WRITEMSG      ; 出错信息
MOV     CX,19            ; 长度
CALL    F10DISPLY        ; 调用显示出错信息程序
MOV     ERRCODE,01       ; 设置错误码
MOV     NAMELEN,00
D20:    RET
D10WRITE ENDP
;
; ----- 关闭磁盘文件: -----
;
E10CLOSE PROC NEAR       ; 使用 AH, BX
MOV     NAMEREC,1AH      ; 设置 EOF
CALL    D10WRITE         ; 标记在文件尾
MOV     AH,3EH           ; 请求关闭磁盘文件
MOV     BX,FILEHAND
INT     21H
RET
E10CLOSE ENDP
;
; ----- 在屏幕上显示数据: -----
;
F10DISPLY PROC NEAR      ; 使用 AX, BX, DX
MOV     AX,1301H         ; BP 和 CX 已设置
MOV     BX,0016H         ; 页属性
MOV     DH,ROW           ; 行
MOV     DL,00            ; 和列
INT     10H
INC     ROW              ; 下一行
RET
F10DISPLY ENDP
END      A10MAIN

```

图 17-2 续

输入区是 30 个字节,紧接着的两个字节是回车符(0DH)和换行符(0AH),总共 32 个字节。程序将 32 字节作为一个固定长度的记录来写入。可以省略回车/换行符,但是如果想对文件中的记录分类则必须包括它们,因为 SORT 程序需要用回车/换行符来表示每个记录的结束。

要注意两点:(1)在每个记录之后包括回车符/换行符只是为了分类更方便,否则可以省略它们。(2)每个记录可以是可变长度的格式,一直到姓名结束,这需要编写一些另外的程序,在后面可以看到。

17.7 读磁盘文件

这一节包括了利用文件代号打开和读磁盘文件的请求。读磁盘文件的过程如下：

- 1. 利用 ASCIIZ 串从系统获取文件代号；
- 2. 利用 INT 21H 的功能 3DH 打开文件；
- 3. 利用 INT 21H 的功能 3FH 从文件中读记录；
- 4. 最后，用 INT 21H 的功能 3EH 关闭文件。

17.7.1 INT 21H 的功能 3DH：打开文件

读文件的程序必须首先用 INT 21H 的功能 3DH 来打开文件。这个操作通过已实际存在的文件名来检查文件，如果存在，对程序是可用的。在 DX 中装入所请求的 ASCIIZ 串的地址，在 AL 中装入一个 8 位的存取代码：

位	请求	位	请求
0-2	000=只读 001=只写 010=读/写	3	1=保留
		4-6	共享方式
		7	继承标志

读文件之前，程序应当用功能 3DH 来打开文件，而不是用功能 3CH 建立文件。下例为读而打开一个文件：

```
FILHAND  DW  ?           ; 文件代号
...
MOV  AH, 3DH           ; 请求打开文件
MOV  AL, 00            ; 存取代码=只读
LEA  DX, PATHNM1       ; ASCIIZ 串
INT  21H               ; 调用中断服务例程
JC   error             ; 出错时的特定动作
MOV  FILHAND, AX       ; 保存文件代号
```

如果指定名字的文件存在，该操作设置记录长度为 1（这点你可以不考虑），确定它的属性，设置文件指针为 0（文件的开始），清除进位标志，在 AX 中返回文件代号。以后对这个文件的所有存取操作都使用这个文件代号。

如果文件不存在，该操作置进位标志为 1，并在 AX 中返回错误码：02、03、04、05 或 12（见图 17-1）。一定要先检查进位标志。例如，建一个文件，正好传送给 AX 的文件代号是 05，这很容易与拒绝存取的错误代码 05 相混淆。程序可以通过滚动屏幕来修改。

17.7.2 INT 21H 的功能 3FH：读记录

INT 21H 的功能 3FH 用于读磁盘记录。在 BX 中放入文件代号，CX 中放入要读的字

节数, DX 中放入输入区的地址。下例用前面的例子得到的文件代号来读一个 512 字节的记录:

```
FILHAND DW ? ; 文件代号
INAREA DB 512 DUP(' ')
...
MOV AH, 3FH ; 请求读记录
MOV BX, FILHAND ; 文件代号
MOV CX, 512 ; 记录长度
LEA DX, INAREA ; 输入区地址
INT 21H ; 调用中断服务例程
JC error ; 出错时的特定动作
CMP AX, 00 ; 读出 0 字节?
JE endfile ; 是, 文件结束
```

一次有效的操作将读出的记录传送给程序, 清除进位标志, 并设置 AX 为实际读出的字节数。AX 为 0 说明试图从文件尾读记录, 这是一个警告而不是一个错误。一次非法的读操作设置进位标志, 并给 AX 返回错误码 05 (拒绝存取) 或 06 (非法的文件代号)。

因为系统限制一次打开的文件数, 所以程序在连续读一定数量文件之后应立即关闭它们。

17.7.3 程序: 顺序读磁盘文件

图 17-3 的程序读取已由图 17-2 的程序建立的文件, 并用 DOS 命令 SORT 来分类。对这个例子, 将 NAMEFILE.DAT 中的记录按升序分类存入 NAMEFILE.SRT, 其命令为:

```
SORT n: <NAMEFILE.DAT >NAMEFILE.SRT
```

(SORT 处理 NAMEFILE.DAT 文件成为 NAMEFILE.SRT)。

下面是主要的过程:

- A10MAIN 调用 B10OPEN、C10READ、D10DISPLY, 在程序末尾关闭文件, 并结束处理。
- B10OPEN 用 INT 21H 的功能 3DH 打开文件, 并保存文件代号。
- C10READ 调用 INT 21H 的功能 3FH, 它利用获取的文件代号来读记录。
- D10DISPLY 显示记录并前移光标。

```
TITLE      A17RDFIL (EXE) 顺序读磁盘文件,
;                               并在屏幕上显示
.MODEL     SMALL
.STACK     64
.DATA
RECD_LEN   EQU      32
ENDCODE    DB        00
FILEHAND   DW        ? ; 结束处理指示符
RECAREA    DB        ? ; 文件代号
RECAREA    DB        32 DUP(' ') ; 记录区
OPENMSG     DB        '*** Open error ***'
PATHNAME    DB        'C:\NAMEFILE.SRT', 0
READMSG     DB        '*** Read error ***'
ROW         DB        00
.386 ;-----
.CODE
```

图 17-3 顺序读记录

```

A10MAIN  PROC  FAR
          MOV  AX,@data      ;初始化
          MOV  DS,AX         ;段寄存器
          MOV  ES,AX         ;寄存器
          MOV  AL,00H        ;清除
          CALL Q10SCROLL     ;全屏
          CALL B10OPEN       ;打开文件
          CMP  ENDCODE,00     ;打开有效?
          JNZ  A90           ;否,退出

A20:      CALL C10READ        ;读磁盘记录
          CMP  ENDCODE,00     ;正常读?
          JNZ  A80           ;否,退出
          LEA  BP,RECAREA     ;是,显示姓名
          MOV  CX,RECD_LEN    ;长度
          CALL D10DISPLY     ;
          JMP  A20           ;继续

A80:      MOV  AH,3EH         ;请求关文件
          MOV  BX,FILEHAND    ;
          INT  21H

A90:      MOV  AX,4C00H       ;结束处理
          INT  21H

A10MAIN  ENDP

;
;      打开文件,测试是否有效:
;
B10OPEN  PROC  NEAR          ;使用 AX, BP, CX, DX
          MOV  AH,3DH         ;请求打开
          MOV  AL,00          ;正常文件
          LEA  DX,PATHNAME
          INT  21H
          JC   B20            ;出错?
          MOV  FILEHAND,AX    ;否,保存文件代号
          JMP  B90

B20:      LEA  BP,OPENMSG     ;是,显示信息
          MOV  CX,18          ;长度
          CALL D10DISPLY     ;出错信息
          MOV  ENDCODE,01     ;是,设置无效标志

B90:      RET

B10OPEN  ENDP

;
;      读磁盘记录,测试是否是文件尾:
;
C10READ  PROC  NEAR          ;使用 AX, BP, BX, CX, DX
          MOV  AH,3FH         ;请求读
          MOV  BX,FILEHAND
          MOV  CX,RECD_LEN    ;32字节为名字和回车/换行
          LEA  DX,RECAREA
          INT  21H
          JC   C20            ;读出错?
          CMP  AX,00          ;文件尾?
          JE   C30
          CMP  RECAREA,1AH    ;文件结束标记?
          JE   C30            ;是,退出
          JMP  C90

C20:      LEA  BP,READMSG     ;否,信息长度
          MOV  CX,18          ;
          CALL D10DISPLY     ;出错信息
          MOV  ENDCODE,01     ;强制结束

C30:      MOV  ENDCODE,01
C90:      RET
C10READ  ENDP

;
;      显示程序,测试屏幕底边:
;
D10DISPLY PROC  NEAR          ;使用 AX, BX, DX
          MOV  AX,1301H       ;BP和CX已设置
          MOV  BX,0016H       ;页属性
          MOV  DH,ROW         ;行
          MOV  DL,10          ;列
          INT  10H
          CMP  ROW,23         ;屏幕底?
          JAE  D80            ;是,跳转
          INC  ROW            ;否,行号增加
          JMP  D90

D80:      MOV  AL,01H         ;滚屏
          CALL Q10SCROLL      ;1行

```

图 17-3 续

```

D90:      RET
DIODISPLY ENDP
;
;          卷屏:
Q10SCROLL PROC    NEAR          ;使用 AX, BH, CX, DX
                MOV    AH,06H    ;AL 已设置
                MOV    BH,1EH    ;设置属性
                MOV    CX,0000
                MOV    DX,184FH  ;请求卷屏
                INT     10H
                RET
Q10SCROLL ENDP
                END      A10MAIN

```

图 17-3 续

17.8 随机处理

前面关于顺序处理磁盘文件的讨论对建立一个文件、打印文件内容以及改变几个小文件已经足够了。然而许多应用要求对一个文件存取特定的记录，如顾客或库存零件的信息。

为了用新的数据修改一个文件，限定顺序处理的程序可能要读出文件中的每个记录，一直到找出所需要的记录为止。例如，为了要存取文件中第 300 个记录，顺序处理可能在传送第 300 个记录之前要读出它前面的 299 个记录（虽然系统可以从指定的记录号开始）。

解决的办法是使用随机处理方法，它可以使一个程序直接存取文件中的任何一个指定的记录。虽然程序顺序地建立了一个文件，但它可以顺序地或随机地存取记录。

当程序随机地请求一个记录时，读操作利用目录来定位记录所在的扇区，并把整个扇区从磁盘读到缓冲区，然后把所需要的记录传送给程序。

在下一个例子中，记录的长度是 128 个字节，一个扇区有 4 个记录。随机请求 21 号记录使下列 4 个记录从扇区中读入缓冲区：

记录 #20	记录 #21	记录 #22	记录 #23
--------	--------	--------	--------

当程序随机地请求下一个记录，如 23 号记录，操作先检查缓冲区。如果记录已经在缓冲区了，它就直接被传送给程序；如果程序请求的记录号不在缓冲区中，则操作又用目录来定位含有这个记录的扇区，并把整个扇区读入缓冲区，然后把这个记录传送给程序。在这种情况下，请求的随机记录号在文件中是靠近的，那么磁盘存取操作也只会很少的几次。

17.8.1 INT 21H 的功能 42H: 移动文件指针

打开操作把文件指针初始化为 0，接着的顺序读和写操作对文件指针增量以处理每个记录。可以使用 42H 功能（移动文件指针）把文件指针设置到文件内的任何地方，然后使用其他的 service 例程随机地检索或更新记录。

为了请求功能 42H，在 BX 中设置文件代号，在 CX:DX 中设置所要求的字节的偏移量。对于最大为 65 535 字节的偏移量，CX 设置为 0，DX 设置为偏移值。还要在 AL 中设置方法

代码，方法代码告诉操作从文件的哪一点开始计算偏移量：

00 从文件开始计算偏移量。

01 从文件指针的当前位置计算偏移量，文件指针可以在文件内的任何地方，包括文件的开始。

02 从文件尾计算偏移量。可以用这个方法编码把记录位置增加到文件尾。或者把 CX:DX 清为 0，然后用方法代码 02 来确定文件的大小。

下例从文件开始把指针移动 1 024 字节：

```

MOV AH, 42H          ; 重定位指针
MOV AL, 00           ; 从文件首开始
MOV BX, HANDLE1      ; 设置文件代号
MOV CX, 00           ; 偏移量高位部分
MOV DX, 1 024        ; 偏移量低位部分
INT 21H              ; 调用中断服务例程
JC error             ; 出错时的特定动作

```

对于有效的操作将清除进位标志，并把新的指针地址传送给 DX:AX。程序可以利用这个地址来执行读或写操作以便随机地处理文件。对于无效的操作，设置进位标志，并在 AX 中返回错误码 01（非法方法代码）或 06（非法文件代号）。

17.8.2 程序：随机读磁盘文件

图 17-4 的程序读取图 17-2 建立的文件。键入文件范围内的一个记录号，就可以请求要在屏幕上显示的任一个文件记录。如果文件包括 24 个记录，那么有效的记录号是 01 到 24。从键盘输入的记录号是 ASCII 格式的，在这种情况下应当只是一位数字或两位数字。

程序的组织如下：

- A10MAIN 调用 B10OPEN、C10RECNO、D10READ 和 E10DISPLY，当用户没有更多的请求时结束。
- B10OPEN 打开文件并获得文件代号。
- C10RECNO 从键盘接收记录号，并检查它在参数表中的长度。在下一节中有详细描述。
- D10READ 利用功能 42H 和相对于 RECINDX 的位置来设置文件指针，并发出 3FH 功能调用，将所要求的记录送入程序的 IOAREA。
- E10DISPLY 显示检索到的记录。

记录号的长度有 3 种可能：00=请求数字的处理结束，01=请求一位数字存入 AL，02=请求 2 位数字存入 AX。子程序必须将 ASCII 数转换为二进制。由于这个数存在 AX 中，所以用 AAD 指令就可处理这个问题，例如，假定输入的记录号是 ASCII 码的 14，那么 AX 中就是 3 134：

- 用 AND 指令将这个值转换为 0104；
- 用 AAD 指令进一步将它转换为 000E(14)；
- 用 DEC 指令减 1（因为系统确认文件是从 0 地址开始）得到 000D（13）；
- 用 SHL 指令左移 5 位，相当于乘以 32（文件中记录的长度），这样得到 1A0（416），

保存在 RECINDX 字段中。

对程序应当进行改进的地方是确认输入的记录号 (01—24)。

```

TITLE      A17RDRAN (EXE) 随机读磁盘记录
.MODEL     SMALL
.STACK     64
.DATA
BOTT_ROW   EQU     22           ; 屏幕底边
RECD_LEN   EQU     32           ; 磁盘记录长度
FILEHAND   DW      ?           ; 文件代号
RECINDEX   DW      ?           ; 记录索引
ERRCODE    DB      00           ; 读错误标记
PROMPT     DB      'Record number? '
RECCAREA   DB      32 DUP(' ') ; 磁盘记录区
PATHNAME    DB      'C:\NAMEFILE.SRT', 0
OPENMSG    DB      '*** Open error ***'
READMSG    DB      '*** Read error ***'
ROW         DB      00
COL         DB      10

RECDPAR    LABEL    BYTE        ; 输入参数表:
MAXLEN     DB      3            ; 最大长度
ACTLEN     DB      ?            ; 实际长度
RECDNO     DB      3 DUP(' ')  ; 记录号
.386 ;
-----
.CODE
A10MAIN    PROC     FAR
MOV        AX,@data            ; 初始化段寄存器
MOV        DS,AX
MOV        ES,AX
MOV        AL,00H              ; 清除全屏
CALL       Q10SCROLL
CALL       B10OPEN             ; 打开文件
CMP        ERRCODE,00          ; 打开有效?
JNZ        A90                 ; 否, 退出

A20:
MOV        COL,10              ; 重置列
CALL       C10RECNO            ; 请求记录#
CMP        ACTLEN,00           ; 还有记录?
JE         A90                 ; 否, 退出
CALL       D10READ             ; 读磁盘记录
CMP        ERRCODE,00          ; 正常读?
JNZ        A90                 ; 否, 退出
LEA        BP,RECCAREA         ; 是, 显示姓名
MOV        CX,RECD_LEN         ; 长度
CALL       E10DISPLY
CMP        ROW,BOTT_ROW        ; 屏幕底?
JAE        A30                 ; 是, 跳转
INC        ROW                 ; 否, 增加行
JMP        A20

A30:
MOV        AL,01H              ; 卷屏
CALL       Q10SCROLL           ; 1行
JMP        A20                 ; 继续

A90:
MOV        AX,4C00H            ; 结束处理
INT        21H
A10MAIN    ENDP

;
; 打开文件, 测试是否有效?
;
B10OPEN    PROC     NEAR
MOV        AX,3D00H            ; 使用 AX, BP, CX, DX
LEA        DX,PATHNAME         ; 请求打开
INT        21H                 ; 正常文件
JC         B20                 ; 出错?
MOV        FILEHAND,AX         ; 否, 保存文件代号
JMP        B90

B20:
MOV        ERRCODE,01          ; 是,
LEA        BP,OPENMSG          ; 显示信息
MOV        CX,18               ; 长度
CALL       E10DISPLY

```

图 17-4 随机读磁盘记录

```

B90:      RET
B10OPEN  ENDP
;
;      取用户指定的记录号, 检查长度, 转换记录号为二进制, 乘32为记录索引:
;
C10RECNO PROC    NEAR                                ; 使用 AX, BP, CX, DX
        LEA    BP, PROMPT                            ; 显示提示信息
        MOV    CX, 15                                ; 长度
        CALL   E10DISPLY
        MOV    AH, 0AH                                ; 请求输入
        LEA    DX, RECDPAR                            ; 记录号
        INT    21H
        CMP    ACTLEN, 01                            ; 检验长度 0, 1, 2
        JB     C40                                    ; 长度 = 0, 结束
        JA     C20                                    ; 长度 = 1 字符
        XOR    AH, AH
        MOV    AL, RECDNO
        JMP    C30

C20:      MOV    AH, RECDNO                            ; 长度 = 2 字符
        MOV    AL, RECDNO+1
C30:      AND    AX, 0F0FH                            ; 清除 ASCII 中的丁
        AAD                                ; 转换为二进制
        DEC    AX                                    ; 调整 (第1个记录为0)
        SHL    AX, 05                                ; 乘32
        MOV    RECIINDEX, AX                        ; 保存记录索引
C40:      MOV    COL, 30
        RET
C10RECNO ENDP
;
;      根据记录索引随机读磁盘记录, 测试是否是有效操作:
;
D10READ  PROC    NEAR                                ; 使用 AX, BP, BX, CX, DX
        MOV    AH, 42H                                ; 请求设置
        MOV    AL, 00                                ; 文件指针到文件首
        MOV    BX, FILEHAND                          ; 文件代号
        MOV    CX, 00                                ; 偏移最高位部分
        MOV    DX, RECIINDEX                        ; 偏移最低位部分
        INT    21H
        JC     D20                                    ; 出错条件?
                                                ; 是, 跳转

        MOV    AH, 3FH                                ; 请求读
        MOV    BX, FILEHAND
        MOV    CX, RECD_LEN                        ; 32 字节为名字和回车/换行
        LEA    DX, RECAREA
        INT    21H
        JC     D20                                    ; 读出错?
        CMP    RECAREA, 1AH                        ; EOF 标记?
        JE     D30                                    ; 是, 退出
        JMP    D90

D20:      ; 否,
        LEA    BP, READMSG                            ; 显示信息
        MOV    CX, 18                                ; 长度
        CALL   E10DISPLY
        MOV    ERRCODE, 01                            ; 强制结束
D30:      RET
D90:      RET
D10READ  ENDP
;
;      显示例程:
;
E10DISPLY PROC    NEAR                                ; 使用 AX, BX, DX
        MOV    AX, 1301H                            ; BP, CX 已输入数据
        MOV    BX, 0016H                            ; 页和属性
        MOV    DH, ROW                                ; 行
        MOV    DL, COL                                ; 列
        INT    10H
        RET
E10DISPLY ENDP
;
;      卷屏, 设置属性:
;

```

图 17-4 续

```

Q10SCROLL PROC    NEAR                                ;使用 BH, CX, DX
                  MOV    AH, 06H                      ;AL 已设置参数
                  MOV    BH, 1EH                      ;设置属性
                  MOV    CX, 0000
                  MOV    DX, 184FH                    ;请求卷屏
                  INT     10H
                  RET
Q10SCROLL ENDP
END      A10MAIN

```

图 17-4 续

17.8.3 程序：读一个 ASCII 文件

前面的例子建立了文件并读取它们，你可能也想处理一个由编辑程序或字处理程序建立的 ASCII 文件。这时必须了解目录和 FAT 的组织以及系统在扇区存储数据的方法。例如，.ASM 文件中的数据严格地按照键入的方式存储，包括 Tab (09H)、回车 (0DH) 和换行 (0AH) 等字符。为了节省磁盘空间，一行上的空格，包括前面讲到的 Tab 符或 Spaces，直到遇到一个回车符，这些在屏幕上直接出现的空格就不存储了。下面是从键盘输入的一条指令：

```
<Tab>MOV<Tab>AH, 09<Enter>
```

对这个 ASCII 数据用十六进制表示为：

```
094D4F560941482C30390D0A
```

这里十六进制的 09 是 Tab，0D 是回车，0A 是换行。当编辑程序或字处理程序读文件时，Tab、回车和换行符自动地调整屏幕上的光标。

现在来分析图 17-5 的程序，它读取并显示文件 A17RDRAN.ASM（图 17-3 的程序），每次读取一个扇区，程序显示每一行直到出现回车/换行符。

- A10MAIN 调用 B10OPEN C10READ 来读第一扇区以及 D10XFER，并在程序最后关闭文件。
- B10OPEN 打开文件，保存文件代号，确定文件的大小（根据 AX 中文件大小的低位部分）。
- C10READ 把一个扇区的数据全部读到 SECTOR。
- D10XFER 从扇区把数据传输到一个显示行，调用 E10DISPLY 来显示，调用 C10READ 读下一个扇区，如此连续处理一直到文件尾。这个过程将在本节后面详细描述。
- E10DISPLY 显示包括换行符在内的显示行的数据，一直显示到换行符。因为 ASCII 文件中的行是可变长度格式的，所以必须在显示之前扫描到每行的行尾（屏幕操作 INT 10H 的功能 13H 对 Tab 符不起作用，但可以用一个循环显示它们）。
- F10ERROR 为磁盘错误显示一条信息。


```

TITLE      A17RDASC (EXE) 读/显示 ASCII 文件
.MODEL     SMALL
.STACK     64
.DATA
DISPAREA  DB      80 DUP(' ')      ; 显示区
ENDCODE   DW      00                ; 结束处理标志
FILESIZE  DW      0                 ; 文件大小 (低位)
FILEHAND  DW      0                 ; 文件代号
OPENMSG   DB      '*** Open error ***'
PATHNAME  DB      'C:\A17RDRAN.ASM', 0
ROW       DB      00
SECTOR    DB      512 DUP(' ')      ; 输入区
;
;-----
.CODE
A10MAIN    PROC     FAR              ; 主程序
MOV        AX,@data                ; 初始化
MOV        DS,AX                    ; 段寄存器
MOV        ES,AX                    ;
MOV        AX,0003H                 ; 设置显示方式和
INT        10H                      ; 清屏
CALL       B10OPEN                  ; 打开文件
CMP        ENDCODE,00               ; 打开有效?
JNE        A90                      ; 否, 退出
CALL       C10READ                  ; 读第一个磁盘扇区
CMP        ENDCODE,00               ; 文件尾, 没有数据?
JE         A90                      ; 是, 退出
CALL       D10XFER                  ; 显示/读
A90:
MOV        AH,3EH                   ; 请求关文件
MOV        BX,FILEHAND
INT        21H
MOV        AX,4C00H                 ; 结束处理
INT        21H
A10MAIN    ENDP
;
;----- 打开磁盘文件, 如有效, 计算文件大小: -----
B10OPEN    PROC     NEAR              ; 使用 AX, BX, CX, DX
MOV        AX,3D00H                 ; 请求打开, 只读
LEA        DX,PATHNAME
INT        21H
JNC        B20                      ; CF=1?
CALL       F10ERROR                 ; 是, 出错
JMP        B90
B20:
MOV        FILEHAND,AX              ; 保存文件代号
MOV        AX,4202H                 ; 设置文件指针
MOV        BX,FILEHAND              ; 到文件尾
MOV        CX,0                     ; 确定
MOV        DX,CX                    ; 文件大小
INT        21H
MOV        FILESIZE,AX              ; 保存文件大小 (低位)
MOV        AX,4200H                 ; 垂直文件指针
MOV        DX,CX                    ; 到文件首
INT        21H
B90:
RET
B10OPEN    ENDP
;
;----- 读磁盘扇区: -----
C10READ    PROC     NEAR              ; 使用 AH, BX, CX, DX
MOV        AH,3FH                   ; 请求读
MOV        BX,FILEHAND              ; 文件代号
MOV        CX,512                   ; 扇区长度
LEA        DX,SECTOR                ; 缓冲区
INT        21H
MOV        ENDCODE,AX               ; 保存状态
RET
C10READ    ENDP
;
;----- 传输数据到显示行: -----
D10XFER    PROC     NEAR              ; 使用 AL, DI, DX, SI
CLD
LEA        SI,SECTOR

```

图 17-5 读一个 ASCII 文件

```

D20:    LEA    DI,DISPAREA
D30:    LEA    DX,SECTOR+512
        CMP    SI,DX                ; 扇区尾?
        JNE    D40                  ; 否, 跳转
        CALL   C10READ              ; 是, 读下一个扇区
        CMP    ENDCODE, 00          ; 文件结束
        JE     D80                  ; 是, 退出
        LEA    SI,SECTOR

D40:    LEA    DX,DISPAREA+60
        CMP    DI,DX                ; 显示区末尾?
        JB     D50                  ; 否, 跳转
        CALL   E10DISPLY            ; 是, 显示
        LEA    DI,DISPAREA

D50:    LODSB                       ; [SI] 到 AL, SI 加1
        STOSB                       ; AL 送入 [DI], DI 加1
        DEC    FILESIZE             ; 全部字符处理完?
        JZ     D80                  ; 是, 退出
        CMP    AL, 0AH              ; 换行?
        JNE    D30                  ; 否, 循环
        CALL   E10DISPLY            ; 是, 显示
        JMP    D20

D80:    CALL   E10DISPLY            ; 显示最后一行
D90:    RET
D10XFER ENDP

;
;                                     显示行, 测试屏幕底边:
;
E10DISPLY PROC NEAR                ; 使用 AX, BP, BX, CX, DX
        MOV    AX,1301H             ; 请求显示
        MOV    BX,0061H             ; 页和属性
        LEA    BP,DISPAREA
        LEA    CX,DISPAREA          ; 计算
        NEG    CX                   ; 行长度
        ADD    CX,DI
        MOV    DH,ROW
        MOV    DL,10
        INT    10H
        CMP    ROW,24               ; 屏幕底?
        JAE    E20                  ; 否, 退出

        INC    ROW                  ; 下一行

E20:    JMP    E90
        MOV    ROW,00
        MOV    AH,10H
        INT    16H                  ; 等待
        MOV    AX,0003H             ; 键盘输入
        INT    10H                 ; 设置显示方式
        RET                          ; 和清屏

E90:    RET
E10DISPLY ENDP

;
;                                     显示磁盘出错信息:
;
F10ERROR PROC NEAR                ; 使用 AX, BP, BX, CX, DX
        MOV    AX,1301H             ; 请求显示
        MOV    BX,0031H             ; 页和属性
        LEA    BP,OPENMSG
        MOV    CX,18                 ; 长度
        MOV    DX,1020H             ; 行: 列
        INT    10H
        MOV    ENDCODE,01           ; 出错标志符
        RET

F10ERROR ENDP
END A10MAIN

```

图 17-5 续

过程 E10DISPLY 从 SECTOR 一次传输一个字节到 DISPAREA, 要显示的字符都存放在这里。必须检查扇区末尾(为了读下一个扇区)并显示区的末尾。对常规的 ASCII 文件, 如 .ASM 文件, 每一行都是相当的短并且也是用回车/换行符来结束。非 ASCII 文件, 如 .EXE 和 .OBJ

文件没有行，那么程序必须检查 DISPARA 的末尾，以避免把数据移进下面的区域。这个程序只设计为显示 ASCII 文件，测试 DISPARA 的末尾是为了防止意外的文件类型出现。检测的步骤如下：

1. 初始化 SECTOR 和 DISPARA 的地址。
2. 如果在 SECTOR 的末尾，则读下一个扇区。如果在文件尾，则退出；否则初始化 SECTOR 的地址。
3. 如果在 DISPARA 的末尾，显示这行并初始化 DISPARA。
4. 从 SECTOR 获取一个字符并把它存入 DISPARA。
5. 如果所有字符都已经处理，退出。
6. 如果字符是换行 (0AH)，显示这行并转向第 2 步；否则转向第 3 步。

尝试在 DEBUG 下对一个适当的驱动器号和 ASCII 文件来运行这个程序。每次磁盘输入之后，显示输入区的内容，并且观察记录是如何被格式化的。加强这个程序的功能，可以提示用户键入文件名和扩展名，并用 DX:AX 来保存文件大小。

17.9 要 点

- 许多 INT 21H 的磁盘服务例程都涉及到一个 ASCIIZ 串，它是由目录路径和紧接着的一个十六进制的 0 字节组成。
- 当出错时，许多磁盘功能设置进位标志，并在 AX 中返回一个错误码。
- 系统为正在处理的程序的每个文件保留一个文件指针。建立和打开操作设置文件指针的值为 0，即文件的开始位置。读和写操作使文件指针前移。
- INT 21H 的功能 3CH 用于在写记录之前建立一个文件，功能 3DH 在读文件之前打开这个文件。这两个操作都返回文件代号，程序在以后存取文件时都要使用这个文件代号。
- 程序在完成写文件操作后，应当关闭文件，这样系统才可以修改目录。
- INT 21H 的功能 42H（移动文件指针）用于随机检索和更新磁盘记录。

17.10 习 题

17-1. 下列情况的错误返回码是什么？(a) 拒绝存取，(b) 文件未发现，(c) 磁盘写保护 (d) 非法功能号

17-2. 为驱动器 C 上的文件 MONITOR.FIL 定义一个名为 ASCSTRING 的 ASCIIZ 串。

17-3. 对 17-2 题的文件编写指令：(a) 建立一个正常属性的文件，如果有效，则保存文件代号，(b) 利用文件代号把一条 100 字节的记录写入文件，(c) 关闭文件。

17-4. 修改图 17-5 的程序，使用户在键盘上可以键入文件名，程序依此来定位文件并显示文件的内容。准备任意个请求并要求只按下回车，程序即开始处理一直到结束。

17-5. 编写程序，允许用户键入零件号（3 个字符）、零件种类（12 个字符）以及单价

(xxx.xx)。程序要建立一个磁盘文件，其中的记录含有这些信息。记住要将单价从 ASCII 码转换为二进制数。下面是输入数据的例子：

零件号	零件名	单价	零件号	零件名	单价
10231	Assemblers	1005251	11221	Lifters	1123201
10241	Linkages	1006301	11241	Processors	1125351
10271	Compilers	1007251	11271	Labelers	1015601
10491	Compressors	1010201	12321	Bailers	1073451
11141	Extractors	1117501	12371	Grinders	1097601
11171	Haulers	1015301	19991		1000001

17-6. 编写程序，显示 17-5 题建立的文件的内容。程序必须将价格的二进制值转换为 ASCII 格式。

17-7. 利用 17-5 题建立的文件按下列要求编写程序：(a) 把记录全部读入到内存的一个表中，(b) 请求一个用户键入零件号和总量，(c) 按零件号搜索表，(d) 如果找到零件号，则用单价表来计算零件的价格（总量×单价），(e) 显示零件名和价格。允许键盘请求任何零件号。

17-8. 一个文件由 100 个记录组成，每个记录的长度为 256 字节。编写指令设置文件指针指向 (a) 文件开始，(b) 第 20 个记录，(c) 文件末尾。

17-9. 编写指令代码，为一个程序确定文件的大小。

17-10. 修改 17-6 题的程序，使它能随机处理磁盘文件。定义有效的零件号及其偏移值的表。请求用户键入零件号，程序在表中查找到这个零件号。用表中的偏移值来计算文件中的偏移值，并用 INT 21H 的功能 42H 来设置文件指针。显示零件名和单价。请求用户输入出售的数量，然后计算并显示出售的总价格（总量×单价）。

磁盘存储 III: 支持磁盘和文件的 INT 21H 功能

目的: 检验支持使用磁盘驱动器和文件处理的各种 INT 21H 操作

18.1 引言

这一章中将介绍一系列的有用操作, 这些内容组织成 3 个部分: 磁盘驱动器处理, 目录和 FAT 处理, 以及磁盘文件处理。在每个部分中, 将以功能码的顺序来描述它们的功能。

处理磁盘驱动器的操作:

- 0DH: 重置磁盘驱动器
- 0EH: 选择默认驱动器
- 19H: 获取默认驱动器
- 1FH: 获取默认驱动器参数块(DPB)
- 2EH: 设置/重置磁盘检验
- 32H: 获取驱动器参数块(DPB)
- 36H: 获取空闲磁盘空间的信息
- 4400H: 获取设备信息
- 4401H: 设置设备信息
- 4404H: 从驱动器中读取控制数据
- 4405H: 写控制数据到驱动器
- 4406H: 检查输入状态
- 4407H: 检查输出状态
- 4408H: 确定设备媒体是否可移动
- 440DH 子功能码 41H: 写磁盘扇区
- 440DH 子功能码 61H: 读磁盘扇区

440DH 子功能码 42H: 格式化磁道
 440DH 子功能码 46H: 设置媒体 ID
 440DH 子功能码 60H: 获取设备的参数
 440DH 子功能码 66H: 获取媒体 ID
 440DH 子功能码 68H: 判断媒体类型
 54H: 获取校验状态
 59H: 获取扩展错误

处理磁盘文件的操作:

1AH: 设置磁盘传输地址
 29H: 解析文件名
 41H: 删除文件
 43H: 获取/设置文件属性
 45H, 46H: 复制文件代号
 4EH, 4FH: 查找匹配文件
 56H: 文件重命名
 57H: 获取/设置文件日期/时间
 5AH, 5BH: 创建临时/新文件

处理目录和 FAT 的操作:

39H: 创建子目录
 3AH: 删除子目录
 3BH: 改变当前目录
 47H: 获取当前目录

本章引用的错误代码参见图 17-1 中的列表。

18.2 处理磁盘驱动器的操作

18.2.1 INT 21H 的功能 0DH: 重置磁盘驱动器

一般来说, 关闭一个文件将导致写入所有剩余记录和更新目录的操作。在特殊环境下, 例如在程序执行步骤之间或者在错误条件下, 程序可以使用功能 0DH 来重置磁盘驱动器:

```
MOV     AH, 0DH      ; 重置磁盘请求
INT     21H          ; 调用中断服务
```

重置操作将清空所有文件缓冲区, 并重置读写头到柱面 0。此操作不关闭文件, 并且无返回值。

18.2.2 INT 21H 的功能 0EH: 选择默认驱动器

0EH 功能的主要目的是选择一个驱动器作为当前默认驱动器。使用时, 在 DL 中设置驱动器号, 0=驱动器 A, 1=驱动器 B, 以此类推:

```
MOV     AH, 0EH      ; 请求设置默认
MOV     DL, 02       ; 驱动器 C
INT     21H          ; 调用中断服务
```

此操作传递驱动器号(所有类型, 包括 RAM 磁盘)到 AL。因为最小的系统至少要求有两个逻辑驱动器 A 和 B, 所以, 对一个单驱动器系统该操作返回的值为 02。(使用 INT 11H 确定实际驱动器号。)

18.2.3 INT 21H 的功能 19H: 获取默认驱动器

此功能确定默认的磁盘驱动器:

```
MOV     AH, 19H      ; 缺省驱动器请求
INT     21H          ; 调用中断服务
```

此操作在 AL 中返回驱动器号, 0=A, 1=B, 以此类推。从默认驱动器访问文件时, 可以直接将驱动器号传送给程序, 尽管有些磁盘操作假定 1=驱动器 A, 2=驱动器 B。

18.2.4 INT 21H 的功能 1FH: 获取默认驱动器参数块(DPB)

驱动器参数块(DPB)是一块数据区, 包含下列驱动器数据结构的底层信息:

偏移量	大小	内容
00H	字节	驱动器号(0=A, 以此类推)
01H	字节	驱动器逻辑单元
02H	字	扇区字节大小
04H	字节	每簇的扇区数减 1
05H	字节	每簇扇区数(2 的幂)
06H	字	FAT 第一个相对扇区
08H	字节	FAT 的拷贝号
09H	字	根目录入口号
0BH	字	第一簇的第一个相对扇区
0DH	字	最高簇号加 1
0FH	字	每个 FAT 占用的扇区
11H	字	目录的第一个相对扇区
13H	双字	安装设备驱动程序地址
17H	字节	媒体描述符

18H	字节	访问标志(0 表示磁盘被访问)
19H	双字	指向下一个参数块的指针
1DH	字	最后分配的簇
1FH	字	空闲簇号

一个有效操作将 AL 清零并且在 DS:BX 中返回一个指向默认驱动器的 DPB 的地址。对错误操作, AL 设置为 FFH。使用此功能前要将 DS 寄存器入栈, 使用 DS 访问 DBP 后要将其出栈。参见功能 32H。

18.2.5 INT 21 的功能 2EH: 设置/重置磁盘验证

这个操作允许程序检验磁盘写操作, 即数据是否正确写入。此操作设置一个开关, 告诉系统检验磁盘控制器的循环冗余校验(CRC, 一种奇偶校验)。在 AL 中装入 00 关闭检验, 01 设置检验。开关的设置一直保持到另一个操作来改变它。例如:

```
MOV     AH, 2EH      ; 请求检验 (或者 MOV  AX, 2E01H)
MOV     AL, 01       ; 设置检验开启
INT     21H          ; 调用中断服务
```

此操作不返回任何值。随后系统回应一个无效写操作。因为磁盘驱动器很少记录错误数据和确认一些延迟原因, 所以这个操作对记录的数据在特别临界的情况下最有用。一个相关的功能是 54H, 它返回校验开关的当前设置。

18.2.6 INT 21H 的功能 32H: 获取驱动器参数块(DPB)

为了获取 DPB, 要在 DX 中装入驱动器号(0=默认, 1=A, 以此类推)。(参见功能 1FH, 除非是要求一个特殊的驱动器, 这一功能是同样的。)

18.2.7 INT 21H 的功能 36H: 获取空闲磁盘空间的信息

此功能传递有关磁盘设备空间的信息。在 DL 中装入驱动盘号(0=默认, 1=A, 2=B, 以此类推):

```
MOV     AH, 36H      ; 请求空闲磁盘空间
MOV     DL, 0         ; 默认驱动器
INT     21H          ; 调用中断服务
```

一个成功的操作返回如下信息: AX=每簇的扇区数; BX=可用簇数; CX=每个扇区字节数; DX=磁盘分区的总簇数。AX, CX 和 DX 的值提供了分区的容量。对于一个无效的设备号, 此操作将在 AX 中返回 FFFFH。这个操作不设置也不清除进位标志。

18.2.8 INT 21H 的功能 44H: 设备的 I/O 控制

这些将要详细阐述的服务 IOCTL 在程序和开放设备之间交流信息。使用这一功能时, 在

AL 中装入子功能参数来请求许多功能中的一种。一个有效操作将进位标志清零。若出错，例如非法文件代号，将设置 CF，并在 AX 中返回一个标准错误代码。主要的 IOCTL 子功能如下所示。

1. INT 21H 的功能 4400H：获取设备信息。这一操作返回关于文件或设备的信息：

```
MOV     AX, 4400H      ; 设备信息要求
MOV     BX, handle     ; 文件或设备的文件代号
INT     21H            ; 调用中断服务
```

有效操作将进位标志清零，并在 DX 中返回一个值，DX 的位 7=0 时表示文件代号指向文件，位 7=1 表示文件代号指向设备。其他位对文件或设备所具有的含义如下：

文件(位 7=0)：

0~5 驱动器号(0=A, 1=B, …)
6 1=文件未写

设备(位 7=1)：

0 标准控制台输入	4 专用设备
1 标准控制台输出	5 0=ASCII 模式, 1=二进制模式
2 空设备	6 对于输入, 0=结束返回的文件
3 时钟设备	

若出错，将设置进位标志，并且在 AX 中返回代码 01, 05 或 06。

2. INT 21H 的功能 4401H：设置设备信息。这一操作设置设备信息，同功能 4400H。使用时，在 BX 中装入文件代号，在 DL 的位 0~7 进行位设置。若出错，将设置 CF，并且在 AX 中返回代码 01, 05, 06 或 0DH。

3. INT 21H 的功能 4404H：从驱动器中读取控制数据。这一操作从块设备驱动器(磁盘驱动器)中读取控制数据。使用时，在 BL 中装入驱动器号(0=默认, 1=A, …)，CX 中为所要读取的字节数，DX 中为数据区的地址。操作成功时，在 AX 中返回的是传输的字节数。若出错，将设置 CF，并且在 AX 中返回代码 01, 05 或 0DH。

4. INT 21H 的功能 4405H：写控制数据到驱动器。这一操作写控制数据到块设备驱动器。除了设置有些不同，其他与功能 4404H 相同。

5. INT 21H 的功能 4406H：检查输入状态。这个服务例程检查文件或设备是否准备好输入。在 BX 中装入文件代号。操作有效时，在 AL 中返回下列中的一个参数：

设备：00H=未准备好, FFH=准备好

文件：00H=到达 EOF, FFH=未到 EOF

若出错，将设置 CF，并且在 AX 中返回 01, 05 或 06。

6. INT 21H 的功能 4407H：检查输出状态。这个服务例程确定一个文件或设备是否准备好输出。操作有效时，在 AL 中返回下列代码：

设备：00H=未准备好, FFH=准备好

文件：00H=未准备好, FFH=准备好

若出错，将设置 CF，并且在 AX 中返回 01, 05 或 06。

7. INT 21H 的功能 4408H: 确定设备媒体是否可移动。这一服务例程确定设备是否包含可移动的媒体, 例如软磁盘。使用时, 在 BX 中装入驱动器号(0=默认, 1=A, ...)。操作成功时, CF 清零, 并且在 AX 中返回如下一个代码: 00H=可移动设备, 01H=固定设备。若出错, 将设置 CF, 并且在 AX 中返回 01 或 0FH(无效驱动器号)。

8. INT 21H 的功能 440DH, 子功能码 41H: 写磁盘扇区。这一操作将缓冲区的数据写入磁盘的一个或多个扇区。使用时, 加载如下寄存器:

```
MOV AX, 440DH      ; 请求写磁盘扇区
MOV BX, drive      ; 驱动器(0=默认, 1=A, ...)
MOV CH, 08H       ; 设备种类=08H
MOV CL, 41H       ; 子功能码=写磁道
LEA DX, devblock   ; 设备块地址
INT 21H           ; 调用中断服务
```

在 DX 中返回的地址以下列格式指向设备块:

```
devblock    LABEL    BYTE      ; 设备块
specfunc    DB        0        ; 特定功能(零)
rwhead      DW        head     ; 读写头
rwcyl      DW        cylinder  ; 柱面
rwsectl     DW        sector   ; 起始扇区
rwsects     DW        number   ; 扇区数
rwbuffer    DW        buffer   ; 缓冲区偏移地址
            DW        SEG _DATA ; 数据段地址
```

入口项 `rwbuffer` 提供了段:偏移(DS:DX)形式的缓冲区地址, 并按字反序存储。SEG 操作符表明定义一个段, 在上例定义的数据段为 `_DATA`。缓冲区表明所要写入的数据区, 缓冲区的长度必须是扇区数 $\times 512$, 例如:

```
WRBUFFER    DB 1024    DUP (?) ; 输出缓冲区(2 扇区 $\times 512$ )
```

操作成功, CF 清零并且写数据。否则, 将设置 CF, 并且在 AX 中返回 01, 02 或 05。

9. INT 21H 的功能 440DH, 子功能码 42H: 格式化磁道。使用这一功能格式化磁道, 设置如下寄存器:

```
MOV AX, 440DH      ; 请求格式化磁道
MOV BX, drive      ; 驱动器(0=默认, 1=A, ...)
MOV CH, 08H       ; 设备种类(08)
MOV CL, 42H       ; 子功能码=格式化磁道
LEA DX, block      ; 块地址(DS:DX)
INT 21H           ; 调用中断服务
```

在 DX 中返回的地址以下列格式指向块:

```
块名        LABEL    BYTE      ; 磁盘信息块:
specfunc    DB        0        ; 特定功能, 代码 0
diskhead     DW        ?        ; 磁盘头
cylinder     DW        ?        ; 柱面
tracks       DW        ?        ; 磁道号
```

操作成功, CF 清零并且格式化磁道。否则, 将设置 CF, 并且在 AX 中返回错误码 01,

02 或 05。

10. INT 21H 的功能 440DH, 子功能码 46H: 设置媒体 ID。使用这一功能设置媒体 ID, 要设置如下寄存器:

```
MOV  AX, 440DH      ; 请求设置媒体 ID
MOV  BX, drive      ; 驱动器 (0=默认, 1=A, ...)
MOV  CH, 08H       ; 设备种类 (08)
MOV  CL, 46H       ; 子功能码=设置媒体 ID
LEA  DX, block      ; 块地址 (DS:DX)
INT  21H           ; 调用中断服务
```

在 DX 中返回的地址以下列格式指向媒体块:

```
blockname    LABEL  BYTE      ; 媒体块:
infolevel    DW      0        ; 信息级别=0
serialno     DD      ??       ; 序列号
vol_label    DB      11 DUP(?) ; 卷标签
filetype     DB      8 DUP(?)  ; FAT 类型
```

入口项 filetype 包含 FAT12 或者 FAT16 的 ASCII 值, 并以空格结尾。操作成功, CF 清零并且设置 ID。否则, 将设置 CF, 并且在 AX 中返回错误码 01, 02, 或者 05。(见 INT 21H 功能 440DH, 子功能码 66H。)

11. INT 21H 的功能 440DH, 子功能码 60H: 获取设备参数。使用这一功能得到设备参数, 设置如下寄存器:

```
MOV  X, 440DH      ; 请求获取设备参数
MOV  BX, drive     ; 驱动器 (0=默认, 1=A, ...)
MOV  CH, 08H       ; 设备种类 (08)
MOV  CL, 60H       ; 子功能码=获取参数
LEA  DX, block     ; 块地址 (DS:DX)
INT  21H           ; 调用中断服务
```

在 DX 中返回的地址以下列格式指向设备参数块(DPB):

```
specfunct    DB  ?      ; 特定函数 (0 或 1)
devicetype   DB  ?      ; 设备类型
devattrib    DW  ?      ; 设备属性
cylinders    DW  ?      ; 柱面数
mediatype    DB  ?      ; 媒体类型
bytesects    DW  ?      ; 每个扇区字节数
seccluster   DB  ?      ; 每簇扇区数
ressectors   DW  ?      ; 保留扇区数
fats         DB  ?      ; FAT 数
rootentry    DW  ?      ; 根目录入口数
sectors      DW  ?      ; 扇区总数
mediadescs   DB  ?      ; 媒体描述符
fatsectors   DW  ?      ; 每个 FAT 扇区数
sectrack     DW  ?      ; 每磁道扇区数
heads        DW  ?      ; 读写头数
hiddensect   DD  ?      ; 隐藏扇区数
exsects      DD  ?      ; 扇区字段=0 的扇区数
```

如果 `specfnct` 字段为 0, 信息是关于驱动器中的默认媒体; 如果是 1, 信息是关于当前媒体的。操作成功, CF 清零并且传递数据。否则, 将设置 CF, 并且在 AX 中返回错误码 01, 02 或 05。

12. INT 21H 的功能 440DH, 子功能码 61H: 读磁盘扇区。此操作从一个或多个扇区中读取数据到磁盘缓冲区。设置 CL 为子功能码 61H; 否则, 操作上的技术细节和写扇区的子功能 41H 相同。后面介绍的图 18-1 的程序将说明这个功能。

13. INT 21H 的功能 440DH, 子功能码 66H: 获取媒体 ID。使用这一功能得到媒体 ID:

```
MOV     AX, 440DH    ; 请求媒体 ID
MOV     BX, drive    ; 驱动器 (0=默认, 1=A, ...)
MOV     CH, 08H      ; 设备种类 (08)
MOV     CL, 66H      ; 子功能码=获取媒体 ID
LEA     DX, block     ; 块地址 (DS:DX)
INT     21H          ; 调用中断服务
```

在 DX 中返回的地址以下列格式指向媒体块:

```
blockname    LABEL    BYTE    ; 媒体块:
infolevel    DW        0      ; 信息级别=0
serialno     DD        ?      ; 序列号
vol_label    DB        11 DUP(?) ; 卷标签
filetype     DB        8 DUP(?) ; FAT 类型
```

成功的操作进位标志清零并设置 ID。filetype 字段包含 FAT12 或者 FAT16 的 ASCII 值, 并用空格结尾。否则, 此操作设置 CF, 并且在 AX 中返回错误码 01, 02 或 05。(参见 INT 21H 的功能 440DH, 子功能码 46H。)

14. INT 21H 的功能 440DH, 子功能码 68H: 判断媒体类型。使用这一功能来请求媒体类型, 设置如下寄存器:

```
MOV     AX, 440DH    ; 请求媒体类型
MOV     BX, drive    ; 驱动器 (0=默认, 1=A, ...)
MOV     CH, 08H      ; 设备种类 (08)
MOV     CL, 68H      ; 子功能码=获得媒体类型
LEA     DX, block     ; 块地址 (DS:DX)
INT     21H          ; 调用中断服务
```

在 DX 中返回的地址指向一个 2 字节媒体块来接收数据:

```
defaultval    DB ?      ; 01 为默认值, 02 为其他
mediatype     DB ?      ; 02=720K, 07=1.44MB, 09=2.88MB
```

操作成功, CF 清零并且设置类型。否则, 将设置 CF, 并且在 AX 中返回错误码 01 或者 05。

功能 44H 的其他 IOCTL 操作, 即与文件共享的相关操作, 这里没有列出。

18.2.9 INT 21H 的功能 54H: 获取校验状态

这一服务例程确定磁盘写校验标志的状态(参见功能 2EH 的设置开关)。此操作在 AL 中

返回 00H 表示校验关闭, 01H 表示校验开启。没有出错条件。

18.2.10 INT 21H 的功能 59H: 获取扩展错误

这一操作提供有关错误的附加信息, 这些错误是在执行 INT 21H 的服务例程, 设置 CF 和 INT 21H 的错误返回码后获得的。该操作返回如下:

AX=扩展错误码	BL=建议动作
BH=错误类别	CH=位置

同时, 该操作 CF 清零, 并且破坏 CL, DI, DS, DX, ES 和 SI 的内容。在中断前请求的所有寄存器进栈, 最后这些寄存器出栈。下面几节要解释这些错误:

AX=扩展错误码。提供大约 90 个或更多的错误码, 00 表示前面的 INT 21H 操作没有错误。

BH=错误类别。提供如下信息:

- 01 出错来源, 例如存储通道
- 02 临时状态(不是错误), 例如一个加锁文件应取消的条件
- 03 缺少适当的授权
- 04 系统软件错, 而非此程序错
- 05 硬件失败
- 06 严重系统错误, 而非此程序错
- 07 此程序出错, 例如不一致请求
- 08 请求条目未找到
- 09 不正确文件或磁盘格式
- 0A 文件或条目上锁
- 0B 磁盘错, 如 CRC 错或者错误盘
- 0C 文件或条目已经存在
- 0D 未知错误类

BL=动作。提供下一步动作的信息:

- 01 重试几次, 可能要求用户结束
- 02 先暂停, 然后重试几次
- 03 要求用户重新输入正确请求
- 04 关闭文件并终止程序
- 05 立即终止程序, 不关闭文件
- 06 忽略错误
- 07 要求用户执行一个动作(如换软盘), 然后重试操作

CH=出错位置。提供定位错误的附加信息:

- 01 未知情况, 无法帮助
- 02 磁盘存储问题
- 03 网络问题

04 串行设备问题

05 存储器问题

18.2.11 程序：从扇区读数据

图 18-1 中的程序说明了 IOCTL 功能 44H 子功能 0DH 的子功能码 61H 的使用。程序从扇区中读数据到内存缓冲区，并且以十六进制字符对的形式显示每个输入字节，如图 15-6 所示。数据段中的块结构 RDBLOCK 任意指定一个读写头、柱面和起始扇区，这可以根据你自己的意图而改变。RDBUFFER 定义了两个地址：

1. IOBUFFER 是输入缓冲区的偏移地址，它提供了一个扇区(512 字节)的数据。
2. SEG_DATA 使用 SEG 操作符为 IOCTL 操作确定数据段地址。

代码段的主要过程是：

- A10MAIN 调用 B10READ，如果操作正确，调用 C10CONVRT。
- B10READ 使用 IOCTL 操作从扇区读数据到 IOBUFFER(测试读是否有效，在程序返回时进行)。
- C10CONVRT 使用 XLAT 指令将 IOBUFFER 中每半个字节转换成一个十六进制字符，并在 DISPAREA 中顺序存储。完成这些工作后，调用 D10DISPLY。
- D10DISPLY 显示 DISPAREA 中的十六进制字符。INT 10H 的功能 13H 在显示到屏幕最右端时自动换行。

一种改进程序的方法是允许用户通过键盘请求任意的起始扇区和扇区数。

```

TITLE      A18RDSCT (EXE) 读磁盘扇区，转换十六进制为
;                                     ASCII并显示
.MODEL     SMALL
.STACK    64
.DATA
XLATAB    DB      30H,31H,32H,33H,34H,35H,36H,37H,38H,39H
           DB      41H,42H,43H,44H,45H,46H
READMSG    DB      '*** Read error ***'

RDBLOCK    DB      0                ; 块
RDHEAD     DW      0                ; 结构
RDCYLDER    DW      0                ;
RDSECTOR    DW      12              ;
RDNOSEC     DW      1                ;
RDBUFFER    DW      IOBUFFER        ;
           DW      SEG DATA        ;
IOBUFFER    DB      512 DUP(' ')    ; 磁盘扇区
DISPAREA    DB      1024 DUP(' ')   ; 显示区
.386 ;
-----
.CODE
A10MAIN    PROC    FAR
MOV        AX,@data                ; 初始化
MOV        DS,AX                  ; 段寄存器
MOV        ES,AX                  ;
MOV        AX,0003H                ; 设置显示方式
INT        10H                    ; 清屏
CALL       B10READ                 ; 得到扇区数据
JC         A80                    ; 如果读无效，跳转
CALL       C10CONVRT               ; 有效，转换
JMP        A90                    ; 并显示
A80:
LEA        BP,READMSG              ; 无效，显示
MOV        CX,18                  ; 错误信息

```

图 18-1 读磁盘扇区

```

        CALL    D10DISPLY
A90:    MOV     AX,4C00H          ; 结束处理
        INT     21H
A10MAIN ENDP
;
; Read contents of disk sector:
; -----
B10READ PROC NEAR                ; 使用 AX, BX, CX, DX
        MOV     AX,440DH        ; 块设备的 IOCTL
        MOV     BX,01           ; 驱动器A
        MOV     CH,08           ; 设备类别
        MOV     CL,61H          ; 读扇区
        LEA     DX,RDBLOCK      ; 块结构地址
        INT     21H
        RET
B10READ ENDP
;
; 扇区数据从十六进制(1字节)转换为ASCII(2字节);
; -----
C10CONVRT PROC NEAR              ; 使用 AL, BP, CX, DI
        LEA     DI,DISPAREA      ; 初始化数据区
        LEA     SI,IOBUFFER      ; 的地址
C20:    MOV     AL,[SI]           ; 得到一个字节
        SHR     AL,04            ; 右移十六进制数字
        LEA     BX,XLATAB        ; 设置表地址
        XLAT    BX               ; 转换十六进制
        STOSB                    ; 存 AL 到显示区
        MOV     AL,[SI]          ; 得到另半个字节
        AND     AL,0FH           ; 清除十六进制数字左边四位
        XLAT    BX               ; 转换十六进制
        STOSB                    ; 保存 AL 到显示区
        INC     SI               ; IOBUFFER地址加一
        LEA     BX,IOBUFFER+512  ;
        CMP     SI,BX            ; IOBUFFER是否结束?
        JB      C20              ; 否, 重复
        LEA     BP,DISPAREA      ; 是, 显示十六进制字符
        MOV     CX,1024
        CALL    D10DISPLY
        RET
C10CONVRT ENDP
;
; Display data:
; -----
; 使用 AX, BX, DX
; BP, CX 已设置
D10DISPLY PROC NEAR
        MOV     AX,1301H        ; 请显示
        MOV     BX,0016H        ; 页: 属性
        MOV     DX,0500H        ; 行: 列
        INT     10H
        RET
D10DISPLY ENDP
        END      A10MAIN

```

图 18-1 续

18.3 处理目录和 FAT 的操作

18.3.1 INT 21H 的功能 39H: 创建子目录

这一服务例程创建子目录, 如同系统命令 MKDIR。使用时, 在 DX 中装入含有驱动器和目录路径名的 ASCII 串的地址:

```

ASCstrg DB 'n:\pathname', 00H    ; ASCII 串
...
MOV  AH, 39H                      ; 请求创建子目录
LEA  DX, ASCstrg                  ; ASCII 串地址(DS:DX)

```

INT 21H ; 调用中断服务

操作有效时 CF 清零, 出错时设置 CF, 并且在 AX 中返回 03 或 05。

18.3.2 INT 21H 的功能 3AH: 删除子目录

这一服务例程删除子目录, 如同系统命令 RMDIR。注意不能删除当前(活动)目录, 或者包含文件的目录。在 DX 中装入含有驱动器和目录路径名的 ASCIIZ 串的地址:

```
ASCstrg  DB  'n:\pathname', 00H ; ASCIIZ 串
--
MOV  AH, 3AH ; 请求删除子目录
LEA  DX, ASCstrg ; ASCIIZ 串地址 (DS:DX)
INT  21H ; 调用中断服务
```

操作有效时 CF 清零, 出错时设置 CF 并且在 AX 中返回 03, 05 或 10H。

18.3.3 INT 21H 的功能 3BH: 改变当前目录

这一服务例程改变当前目录到指定目录, 如同系统命令 CHDIR。使用时, 在 DX 中装入含有驱动器和目录路径名的 ASCIIZ 串的地址:

```
ASCstrg  DB  'n:\pathname', 00H ; ASCIIZ 串
--
MOV  AH, 3BH ; 请求改变目录
LEA  DX, ASCstrg ; ASCIIZ 串地址 (DS:DX)
INT  21H ; 调用中断服务
```

操作有效时 CF 清零, 出错时设置 CF 并且在 AX 中返回 03。

18.3.4 INT 21H 功能的 47H: 获取当前目录

这一服务例程可以得到任意驱动器的当前目录。使用时, 定义一块缓冲区空间, 其大小足以能容纳可能最长的路径名(64 字节), 并且将其地址存入 SI。在 DL 中装入驱动器号(0=默认, 1=A, 2=B, ...):

```
buffer  DB  64  DUP(20H) ; 64 字节缓冲区空间
--
MOV  AH, 47H ; 请求获得目录
MOV  DL, drive ; 驱动器号
LEA  SI, buffer ; 缓冲区地址 (DS:SI)
INT  21H ; 调用中断服务
```

操作有效时 CF 清零, 并且将当前目录(不是驱动器)的名字以 ASCIIZ 串的形式传送到缓冲区, 如 PCPROGS\TESTDATA, 后接 1 字节 00H 表示路径名结束。如果请求的目录是根目录, 返回值只是 1 字节的 00H。这样, 可以得到当前目录名以便访问子目录下的任何文件。一个无效的驱动器号将设置 CF, 并且在 AX 中返回错误码 0FH。

18.3.5 INT 21H 的功能 56H: 文件或目录重命名

此功能见下一节。

18.3.6 程序: 显示目录

图 18-2 中的程序说明了前节所描述的两个功能的用法。程序的目的是显示默认的目录。程序执行以下功能:

- 使用功能 19H 在 AL 中得到默认驱动器, 返回驱动器号为 0(驱动器 A), 1(驱动器 B), 依此类推。为了将数字调整为相应的字母, 程序中简单地加上了 41H, 这样 00 变为 41H(A), 01 变为 42H(B), 依此类推。
- 在 DRIVE 字段中存储驱动器对应的字母, 后接冒号和斜线(n:\)。
- 使用功能 47H 得到当前目录的路径名, 存储在 PATHNAME 中。
- 扫描查找 ASCII 串的终止符 00H, 以确定 PATHNAME 中路径名的长度。
- 使用这一长度显示 DRIVE 和 PATHNAME。
- 结束前等待键盘输入。

PATH_LEN	EQU	64	
DRIVE	DB	' ', ': \'	; 驱动器
PATHNAME	DB	64 DUP(' ')	; 当前路径名
	...		
	MOV	AH, 19H	; 请求默认驱动器
	INT	21H	
	ADD	AL, 41H	; 转换十进制数为字母
	MOV	DRIVE, AL	; 0=A, 1=B, 等
	MOV	AH, 47H	; 请求默认
	MOV	DL, 00	; 驱动器
	LEA	SI, PATHNAME	; 的路径名
A20:	INT	21H	
	MOV	AL, 00H	; 扫描
	MOV	CX, PATH_LEN	; PATHNAME
	LEA	DI, PATHNAME	; 找 00H
	REPNE	SCASB	
	CMP	CX, PATH_LEN	; 没有终止符?
	JE	A90	; 有, 退出
	NEG	CX	; 符号求反
	ADD	CX, PATH_LEN+2	; 路径名长度
	MOV	AX, 1301H	; 显示路径名
	MOV	BX, 0016H	; 页: 属性
	LEA	BP, DRIVE	; 开始地址
	MOV	DX, 0A0CH	; 行: 列
	INT	10H	
A90:	MOV	AH, 10H	; 等待用户
	INT	16H	; 按键
	...		

图 18-2 获取当前目录

18.4 处理磁盘文件的操作

这一节描述处理磁盘文件的操作。

18.4.1 INT 21H 的功能 1AH: 设置磁盘传输地址

磁盘传输地址(DTA)是一个简单的存储区, 用来从磁盘接收数据。这一操作主要和功能 4EH 和 4FH 一起使用, 如下例:

```
DTA_ADDRESS DB nn DUP(?)
--
MOV AH, 1AH          ; 请求设置 DTA
LEA DX, DTA_address  ; DTA 地址
INT 21H              ; 调用中断服务
```

18.4.2 INT 21H 的功能 29H: 解析文件名

这一服务例程将一个形式为 `n:filename.ext` 的文件说明(filespec)的命令行转换成 FCB 格式。FCB(File Control Block)是一种不再使用的磁盘存取方法, 它已被文件代号法取代。需要知道的只是 FCB 是文件说明的格式, 它由 11 个字节组成:

1~8 文件名。文件的名称, 左对齐, 其余用空格补齐。

9~11 文件扩展名。左对齐, 其余用空格补齐。

此功能可以从用户接受一个文件说明, 例如复制和删除文件。使用时, 将要解析的文件说明的地址装入 SI(用于 DS:SI), 操作要产生的 FCB 格式的存储区的地址装入 DI(用于 ES:DI), AL 中各位的值用来控制解析方法:

```
MOV AH, 29H          ; 请求解析文件名
MOV AL, code          ; 解析方法
LEA DI, FCBname        ; FCB 地址 (ES:DI)
LEA SI, filespec        ; 文件说明的地址 (DS:SI)
INT 21H              ; 调用中断服务
```

解析方法的代码从右到左是:

位	值	动作
0	0	表明文件说明从第一字节开始。
0	1	扫描分隔符(如空格)查找文件说明。
1	0	在生成的 FCB 中设置驱动器 ID 字节: 未找到驱动器=00, A=01, B=02, 以此类推。
1	1	仅当解析的文件说明指定一个驱动器时, 才在生成的 FCB 中改变驱动器 ID 字节。这种情况下, 一个 FCB 可能有它自己默认的驱动器。
2	0	需要时改变 FCB 中的文件名。
2	1	仅当文件说明中包含有效文件名时, 才改变 FCB 的文件名。
3	0	需要时改变文件扩展名。
3	1	仅当文件说明中包含有效扩展名时, 才改变 FCB 的扩展名。
4~7	0	必须为零。

对于有效数据, 功能 29H 为文件名和扩展名创建一个标准 FCB 格式, 即 8 字符的文件

名, 如有必要用空白符填写, 以及 3 字符的扩展名, 如有必要用空白符填写, 它们之间没有点(.)。

操作能够识别标准的标点符号, 并将通配符*和?转换成一个或多个字符的串。例如, PROG12.*转换为 PROG12bb???(文件名后填入 2 个空白符, ???作为扩展名)。AL 返回下列代码之一: 00H=未遇到通配符; 01H=已转换通配符; FFH=指定非法的驱动器。

操作结束后, DS:SI 含有文件说明解析后第一个字节的地址, ES:DI 含有 FCB 第一个字节的地址。对一次失败的操作, DI+1 的字节为空白符, 尽管操作试图转换你输入的任何代码。

为了用文件代号也能使此操作工作, 需要进一步编辑, 包括删去空白符, 并且在文件名和扩展名之间插入句点。

18.4.3 INT 21H 的功能 41H: 删除文件

此功能从一个程序内删除文件(不包括只读文件)。在 DX 中装入 ASCIIZ 串的地址, ASCIIZ 串包含设备路径和文件名, 不带有通配符标记:

```

ASCStrng  DB  'n:\pathname', 00H      ; ASCIIZ 串
...
MOV  AH, 41H                          ; 请求删除文件
LEA  DX, ASCStrng                      ; ASCIIZ 串地址 (DS:DX)
INT  21H                              ; 调用中断服务

```

有效操作清零 CF, 目录中的文件名标记为已删除, 释放 FAT 中分配给该文件的磁盘空间。操作出错, 设置 CF, 并且在 AX 中返回 02, 03 或 05。

18.4.4 INT 21H 的功能 43H: 获得/设置文件属性

使用此项功能既可以获取也可以设置文件属性。操作要求 ASCIIZ 串的地址装入 DX, ASCIIZ 串包含要操作文件的驱动器、路径和文件名(如果路径未给出, 则使用默认目录)。

1. 获得文件属性。为了得到文件属性, 在 AL 中装入代码 00, 如下例所示:

```

ASCStrng  DB  'n:\pathname', 00H      ; ASCIIZ 串
...
MOV  AH, 43H                          ; 请求
MOV  AL, 00                          ; 获得属性
LEA  DX, ASCStrng                      ; ASCIIZ 串地址 (DS:DX)
INT  21H                              ; 调用中断服务

```

有效操作 CF 清零, CH 清零, 并在 CL 中返回当前属性:

位	属性
0	只读文件
1	隐藏文件
2	系统文件
3	卷标

4 子目录

5 存档文件

操作出错则设置 CF 并且在 AX 中返回代码 02 或 03。

2. 设置文件属性。为了设置文件属性, 在 AL 中装入代码 01, 在 CX 中装入属性。可以改变只读、隐藏、系统和存档属性, 但不可以改变卷标和子目录。下例设置一个文件为隐藏和存档属性:

```
MOV     AH, 43H           ; 请求
MOV     AL, 01           ; 设置属性
MOV     CX, 22H           ; 隐藏和存档属性
LEA     DX, ASCIIZ string ; ASCII 串地址 (DS:DX)
INT     21H              ; 调用中断服务
```

有效操作 CF 清零, 并且在 CX 中设置目录入口项的属性。操作出错, 设置 CF 并且在 AX 中返回代码 02, 03 或 05。

18.4.5 INT 21H 的功能 45H: 复制文件代号

此服务例程的目的是给一个文件多个文件代号。新旧文件代号的使用是相同的, 这些代号指向同一个文件、文件指针和缓冲区。一种用法是请求一个文件代号, 并且用这个文件代号去关闭文件。这一操作导致系统清空缓冲区并更新目录。接着可以使用最初的文件代号继续处理文件。下面是功能 45H 的例子:

```
MOV     AH, 45H           ; 请求复制文件代号
MOV     BX, handle        ; 要复制的当前文件代号
INT     21H              ; 调用中断服务
```

有效操作使 CF 清零, 并且在 AX 中返回下一个可用的文件代号。操作出错, 设置 CF 并且在 AX 中返回错误代码 04 或 06(参见功能 46H)。

18.4.6 INT 21H 的功能 46H: 强迫复制文件代号

此服务例程除了可以分配一个指定的文件代号外, 其他与功能 45H 类似。使用这一功能可以使输出重定向, 例如, 到另外一个路径。使用时, 在 BX 中装入初始的文件代号, CX 中装入第二个文件代号。操作成功 CF 清零。操作出错, 设置 CF 并且在 AX 中返回错误码 04 或 06。有些组合可能不工作, 例如: 代号 00 总是键盘输入, 04 是打印机输出, 03(辅助设备)不能重定向(见功能 45H)。

18.4.7 INT 21H 的功能 4EH: 查找第一个匹配文件

使用 4EH 功能开始查找目录中的第一个相关文件, 使用 4FH 功能继续查找同组的下一个相关文件。需要为操作返回的定位目录入口定义一个 43 字节的缓冲区, 并且在使用这个服务之前使用功能 1AH(设置 DTA)。DTA(Disk Transfer Address)是内存中的一个简单区域, 定义

这个区域用来从磁盘接收数据。设置 DTA 告诉功能 4EH 要求的数据将传送到何处。

开始查找时，在 CX 设置将要返回的文件名的属性——只读(位 0)，隐藏(位 1)，系统(位 2)，卷标(位 3)，目录(位 4)或存档(位 5)，它们可以任意组合。在 DX 中装入包含文件说明的 ASCIIZ 串地址，串中可以包含通配符?和*。例如，对文件说明 n:\ASMPROGS\A12*.ASM 的查找请求将导致操作从与串匹配的第一个文件开始查找。如下例：

```

ASCString  DB  'ASCIIZ string', 00H    ; ASCIIZ 串
...
MOV  AH, 4EH                ; 请求第一次匹配
MOV  CX, 00                  ; 正常属性
LEA  DX, ASCString           ; ASCIIZ 串地址(DS:DX)
INT  21H                     ; 调用中断服务

```

定位一个匹配文件的操作清除进位标志，并且用下列数据填入 43 字节(2BH)的 DTA：

FILEDTA	LABEL	BYTE	;	文件 DTA
	DB	21 DUP(20H)	;	保留为子目录搜索
FILEATTR	DB	0	;	文件属性
FILETIME	DW	0	;	文件时间
FILEDATE	DW	0	;	文件日期
LOWSIZE	DW	0	;	文件大小：低位字
HIGHSIZE	DW	0	;	文件大小：高位字
FILENAME	DB	13 DUP(20H)	;	ASCIIZ 串形式的文件名和扩展名，
			;	紧接十六进制 00

操作出错，设置 CF 并且返回代码 02，03 或 12H。如果打算接着使用功能 4FH，不要改变 DTA 的内容。

功能 4EH 的惟一用处是能够确定查询的是文件名还是子目录。例如，如果返回的属性是 10H，则涉及到的是子目录。操作同时也返回文件的大小，这在图 23-3 中说明。可以使用功能 4EH 确定文件的大小，使用功能 36H 检查写文件的可用空间。

18.4.8 INT 21H 的功能 4FH：查找下一个匹配文件

使用这一服务例程之前，调用功能 4EH 开始在一个目录中查找，然后使用功能 4FH 继续查找：

```

MOV  AH, 4FH                ; 请求下一个匹配文件
INT  21H                     ; 调用中断服务

```

操作成功则 CF 清零并且在 AX 中返回代码 00(找到文件名)或 18(没有更多的文件)。操作出错，设置 CF，并在 AX 中返回错误码 02，03 或 12H。

功能 4EH 和 4FH 将在图 18-3 中说明。

18.4.9 INT 21H 的功能 56H：重命名文件或目录

这一服务例程在一个程序内重新命名文件或者目录。在 DX 中装入 ASCIIZ 串地址，ASCIIZ 串包含原来的驱动器、路径和要重命名的文件名或路径名；在 DI(组成 ES:DI)中装入

另一个 ASCIIZ 串地址, 这个 ASCIIZ 串包含新的驱动器、路径和文件名, 不带通配符。如果使用了驱动器号, 在两个串中必须相同。因为路径不需要相同, 所以操作可以重命名一个文件, 并把它移动到同一驱动器的另一个目录下:

```
oldstring DB 'n:\oldpath\oldname', 00H
newstring DB 'n:\newpath\newname', 00H
--
MOV AH, 56H          ; 请求重命名文件/目录
LEA DX, oldstring    ; DS:DX
LEA DI, newstring    ; ES:DI
INT 21H              ; 调用中断服务
```

操作成功则清除进位标志; 操作出错, 设置 CF, 并在 AX 中返回错误码 02, 03, 05 或 11H。

18.4.10 INT 21H 的功能 57H: 获取/设置文件日期和时间

这一服务例程能使程序获得或设置一个打开文件的时间和日期。时间和日期的形式如下:

表示时间的位		表示日期的位	
0BH~0FH	小时	09H~0FH	年(相对于 1980)
05H~0AH	分	05H~08H	月
00H~04H	秒	00H~04H	日

秒是以每两秒增量的 0-29 之间的数字形式表示。在 AL 中装入请求号(0=获取, 1=设置), 在 BX 中装入文件代号。在请求设置日期/时间时, 在 CX 中装入时间, DX 中装入日期。下面是一个例子:

```
MOV AH, 57H          ; 请求设置
MOV AL, C1           ; 文件的日期和时间
MOV BX, handle       ; 文件代号
MOV CX, time         ; 新的时间(小时|分|秒)
MOV DX, date         ; 新的日期(年|月|日)
INT 21H              ; 调用中断服务
```

有效操作清除进位标志; 获取操作在 CX 中返回时间, DX 中返回日期, 而设置操作改变文件的时间和日期的条目。无效操作设置 CF 并且在 AX 中返回错误码 01 或 06。

18.4.11 INT 21H 的功能 5AH: 创建临时文件

创建临时文件的程序可以使用这个服务例程, 特别是在网络中, 其他文件的名称可能并不知道, 而要求程序避免意外地重写这些文件。此操作在路径中创建一个不重名的文件。

在 CX 中装入所要求的文件属性——只读(位 0), 隐藏(位 1), 系统(位 2), 卷标(位 3), 目录(位 4)或存档(位 5), 它们可任意组合。在 DX 中装入 ASCIIZ 路径的地址——驱动器(如有必要), 子目录(如果存在), 反斜线和 00H, 后跟 13 个空字节存储新文件名:

```
ASCpath DB 'n:\pathname\ ', 00H, 13 DUP(20H)
```

```

MOV AH, 5AH          ; 请求创建文件
MOV CX, attribute    ; 文件属性
LEA DX, ASCpath      ; ASCIIZ 串路径
INT 21H              ; 调用中断服务

```

操作成功, 清除进位标志, 传递文件代号到 AX, 添加新文件名到一个以 00H 字节开头的 ASCIIZ 串。无效的操作设置 CF, 并在 AX 中返回代码 03, 04 或 05。

18.4.12 INT 21H 的功能 5BH: 创建新文件

此服务例程只创建一个文件名已经不存在的文件, 其他方面和功能 3CH (创建文件) 相同。当不想重写一个已存在的文件时, 可以使用功能 5BH。有效操作清除进位标志, 返回文件代号到 AX, 无效操作(包括找到同名的文件), 设置 CF 并且在 AX 中返回代码 03, 04, 05 或 50H。

18.4.13 程序: 选择性删除文件

图 18-3 的程序说明了使用功能 4EH 和 4FH 寻找目录中所有的文件名, 并且使用功能 41H 删除选择的文件。程序假定驱动器为 A, 并由如下过程组成:

- A10MAIN 调用 B10FIRST、C10NEXT、D10MESSG 和 E10DELETE。
- B10FIRST 为调用功能 4EH 设置 DTA, 并且在目录中找到第一个匹配的入口。
- C10NEXT 寻找目录中下一个匹配的入口。
- D10MESSG 显示文件名, 并询问是否要删除它们。
- E10DELETE 接受回答 Y(yes)删除文件, N(no)保留文件, 或者回车结束处理, 若有请求则删除文件。

作为一种预防措施, 可以在测试时使用临时拷贝文件。可以通过添加屏幕滚动来改进程序。

```

TITLE      A18SELDL (EXE) 选择并清除文件
CODESEG
SEGMENT PARA 'Code'
.MODEL SMALL
.STACK 64
.DATA
MMSG_LEN  EQU 32
ROW        DB 00
COL        DB 10
PATHNAME   DB 'A:\*.*', 00H
DELMSG     DB 'Delete? '
ENDMSG     DB 'No more directory entries '
ERRMSG1    DB 'Invalid path/file '
ERRMSG2    DB 'Write-protected disk '
PROMPT     DB 'Y = Delete, N = Keep, Ent = Exit'
DISKAREA   DB 43 DUP(20H)
/
.CODE
A10MAIN    PROC NEAR
MOV        AX, @data
MOV        DS, AX
MOV        ES, AX
MOV        AX, 0003H

```

图 18-3 选择性删除文件

```

INT      10H          ; 并清屏
CALL     B10FIRST      ; 目录入口
CMP      AX, 00H       ; 无入口,
JNE      A90           ; 退出
MOV      CX, MSSG_LEN  ; 提示符长度
LEA      BP, PROMPT    ; 显示初始提示
CALL     F10DISPLY
INC      ROW

A20:
CALL     D10MESSG      ; 显示文件名
CALL     E10DELETE     ; 若需求则删除
CMP      AL, 0FFH      ; 要求结束?
JE       A90           ; 是, 结束
INC      ROW           ; 设置下一行
CALL     C10NEXT       ; 得到下一个目录入口
CMP      AX, 00H       ; 有更多入口?
JE       A20           ; 是, 继续
A90:
MOV      AX, 4C00H     ; 否, 结束处理
INT      21H

A10MAIN  ENDP
;
; 寻找目录中的第一个入口:
; -----
B10FIRST PROC NEAR      ; 使用 AX, BP, CX, DX
MOV      AH, 1AH       ; 为功能调用得到 DTA
LEA      DX, DISKAREA
INT      21H
MOV      AE, 4EH       ; 定位第一个目录入口
MOV      CX, 00
LEA      DX, PATHNAME  ; ASCII 串地址
INT      21H
JNC      B90           ; 有效操作?
PUSH     AX            ; 否, 显示
MOV      CX, MSSG_LEN  ; 结束信息
LEA      BP, ERRMSG1
CALL     F10DISPLY
POP      AX

B90:
RET
B10FIRST ENDP
;
; 寻找目录中下一个入口:
; -----
C10NEXT  PROC NEAR      ; 使用 AX, BP, CX
MOV      AH, 4FH       ; 读下一个目录入口
INT      21H
CMP      AX, 00H       ; 有更多入口?
JE       C90           ; 是, 跳过
PUSH     AX            ; 否, 显示
MOV      CX, MSSG_LEN  ; 结束信息
LEA      BP, ENDMMSG
CALL     F10DISPLY
POP      AX

C90:
RET
C10NEXT  ENDP
;
; 计算文件名长度并显示:
; -----
D10MESSG PROC NEAR      ; 使用 AL, BP, CX, DI
MOV      COL, 10
MOV      CX, 08        ; 信息长度
LEA      BP, DELMSG     ; 显示删除信息
CALL     F10DISPLY
MOV      COL, 18
LEA      DI, DISKAREA+30 ; DTA 中的文件名
MOV      AL, 00H        ; 在磁盘区扫描 13 字节
MOV      CX, 13
REPNE SCASB
NEG      CX              ; 计算文件名长度
ADD      BP, DISKAREA+1EH ; 显示
CALL     F10DISPLY
RET
D10MESSG ENDP
;
; 如有请求则删除记录:

```

图 18-3 续


```

;
E10DELETE PROC NEAR ;使用 AX, BP, CX, DX
MOV AH, 10H ;接收回应字符
INT 16H ; (y/n)
CMP AL, 0DH ;回车符?
JE E80 ;是,退出
OR AL, 00100000B ;变为小写
CMP AL, 'y' ;删除要求?
JNE E90 ;否,跳过MOV AH, 41H
MOV AH, 41H
LEA DX, DISKAREA+1EH ;文件名地址
INT 21H ;删除入口
JNC E90 ;有效删除?
MOV CX, MSSG_LEN ;否,显示警告信息
LEA BP, ERRMSG2 ;
CALL F10DISPLY ;
E80: MOV AL, 0FFH ;结束处理标志
E90: RET
E10DELETE ENDP

;
; 显示程序:
;
F10DISPLY PROC NEAR ;使用 AX, BX, DX
MOV AX, 1301H ;BP, CX 设置
MOV BX, 0016H ;请求显示行
MOV DH, ROW ;页:属性
MOV DL, COL
INT 10H
RET
F10DISPLY ENDP
END A10MAIN

```

图 18-3 续

18.5 要 点

- 有关处理磁盘驱动器的操作包括对设备的重置、选择默认、获取驱动器信息、获取空闲磁盘空间以及设备的扩展操作 I/O 控制。
- 有关处理路径和 FAT 的操作包括创建子目录、删除子目录、改变当前目录和获得当前目录。
- 有关处理磁盘文件(除了创建、打开、读和写)包括文件重命名、获得/设置文件属性、寻找匹配文件和获取/设置日期/时间。

18.6 习 题

使用 DEBUG(或其他调试程序)测试下列问题。键入 A 100 命令和所需要的汇编指令。检查寄存器中返回的值。

18-1. 检验下列有关磁盘驱动器的问题:

- (a) 功能 19H 确定当前默认驱动器。
- (b) 功能 1FH 获取有关默认 DPB 的信息。
- (c) 功能 36H 确定空闲磁盘空间的数量。

- (d) 功能 4400H 获取使用中的设备的信息。
- (e) 功能 4406H 检查输入状态。
- (f) 功能 4408H 确定使用的媒体是否可移动。
- (g) 功能 440DH 子功能码 60H 获得设备参数。
- (h) 功能 440DH 子功能码 66H 获得媒体 ID。

18-2. 检验下列关于目录的问题:

(a) 功能 39H 创建子目录。安全起见, 可以创建在软盘上或者 RAM 盘上。使用任意合法的名字。

(b) 功能 56H 重命名子目录。

(c) 功能 3AH 删除子目录。

18-3. 检验下列有关磁盘文件的问题(在练习中使用文件复本):

(a) 功能 43H 从软盘上的文件获取属性。

(b) 功能 56H 重命名文件。

(c) 功能 43H 设置属性为隐藏。

(d) 功能 57H 得到文件日期和时间。

(e) 功能 41H 删除文件。

18-4. 在 DEBUG 下写一个小程序, 简单地执行 INT 21H 的功能 29H(解析文件名)。在 81H 提供文件说明, FCB 在 5CH, 这些单元都在紧接着程序前面的 PSP。输入各种文件说明, 如 n:ASMPRO1.DOC, ASMPRO2, ASMPRO3.* 和 n:*.ASM。每个解析功能执行后, 在偏移地址 5CH 单元检查结果。

磁盘存储 IV:

INT 13H 磁盘功能

目的：检验使用 BIOS INT 13H 功能来格式化、检验、读磁盘和写磁盘的基本要求。

19.1 引言

在第 17 和 18 章中，我们检验了 INT 21H 的磁盘处理服务例程，也可以使用 INT 13H 在 BIOS 级直接处理，尽管 BIOS 不提供自动使用 FAT、目录或记录的分块和解决功能。BIOS 磁盘操作 INT 13H 处理一个扇区大小的数据，按实际磁道和扇区号来处理磁盘寻址。INT 13H 磁盘操作包括重置、读取、写入、检测和格式化驱动器。

大多数 INT 13H 操作是为有经验的软件开发者使用的，他们了解错误操作带来的潜在危险。同时，由于使用的处理器，甚至计算机的型号不同，BIOS 版本也可能不同。

本章将讲述 INT 13H 的下述功能：

00H	重置硬盘/软盘系统	0CH	搜索柱面
01H	读硬盘/软盘状态	0DH	代替磁盘重置
02H	读扇区	0EH	读扇区缓冲区
03H	写扇区	0FH	写扇区缓冲区
04H	检测扇区	15H	获取硬盘/软盘类型
05H	格式化磁道	16H	改变软盘状态
08H	获取驱动器参数	17H	设置软盘类型
09H	初始化驱动器	18H	设置格式化的媒体类型
0AH	读扩展扇区缓冲区	19H	停置磁头
0BH	写扩展扇区缓冲区		

19.2 BIOS 状态字节

大多数 INT 13H 功能在成功或失败时清除或设置 CF，并且在 AH 寄存器中返回一个状态码。BIOS 在它的数据库保留每个设备及其状态的信息。如图 19-1 所示的状态字节反映了

BIOS 数据区中的指示器各位的意义, 40:41H 是软盘驱动器数据区, 40:74H 是硬盘数据区(详见第 24 章)。

如果磁盘操作返回一个错误, 程序通常的做法是重置磁盘(功能 00H), 并且重试这个操作 3 遍。如果错误仍然出现, 程序将显示信息, 并给用户更换软盘的机会(如果这样可以解决问题的话)。

代码	状 态
00H	无错
01H	错误命令, 未被控制器识别
02H	磁盘上的地址标记未找到
03H	试图写保护盘
04H	无效磁道/扇区
05H	重置操作失败
06H	上次访问后软盘移动
07H	驱动器参数错
08H	直接存储器存取(DMA)过速(存取数据太快, 无法输入)
09H	DMA 超过 64K 的限制, 试图读/写
10H	读盘时遇到坏 CRC(指出被破坏数据的错误检查)
20H	控制器失败(硬件出错)
40H	查找操作失败(硬件出错)
80H	设备无响应(软盘: 驱动器门打开或无软盘; 硬盘: 超时)
AAH	驱动器未准备好
BBH	未定义错
CCH	写失败

图 19-1 INT 13H 状态代码

19.3 基本的 INT 13H 磁盘操作

这一节包含基本的 INT 13H 磁盘操作, 每个请求的功能码都存在 AH 中。

19.3.1 INT 13H 的功能 00H: 重置磁盘系统

前面讲到的磁盘操作在报告了一个严重错误之后, 程序可能使用这个操作。本操作对软盘或者硬盘控制器执行强制性的重置操作, 即下一次访问磁盘时, 它首先重置于柱面 0。对于软盘, 设置 DL 为驱动器号(0=驱动器 A, 等等), 对于硬盘, 设置 DL 为 80H 或大于 80H 的值(80H=驱动器 C, 81H=D, 等等)。使用功能 00H 的例子如下:

```
MOV    AH, 00H        ; 请求重置磁盘
MOV    DL, 80H        ; 磁盘驱动器 C
INT     13H           ; 调用中断服务
```

有效操作清除 CF; 错误操作设置 CF, 并在 AH 中返回状态码。与此相关的操作是功能 0DH。

19.3.2 INT 13H 的功能 01H: 读磁盘状态

此操作提供了检查最近一次磁盘操作状态的另一个机会(见图 19-1 的 BIOS 状态字节)。对于软盘, 设置 DL 为通用码(0=驱动器 A, 等等), 对于硬盘, 设置 DL 为 80H 或大于 80H 的值(80H=第一个驱动器, 等等)。此操作在 AL 中返回状态码, 这个状态码在最近一个磁盘操作时已经返回给 AH 了。这一操作总是有效的, 清除 CF 并且在 AH 中返回它自己的状态码 00H。

19.3.3 INT 13H 的功能 02H: 读扇区

这一操作将同一磁道上的指定数量的扇区直接读取到存储器。注意: 柱面/磁道号从 0 开始, 而扇区号从 1 开始。初始化如下寄存器:

AL	扇区数, 最多为一个磁道的扇区数
CH	柱面/磁道号(低 8 位)
CL	位 7~6 是柱面/磁道号(高 2 位) 位 5~0 为起始扇区号
DH	读写头/盘面号(对于软盘是 0 或 1)
DL	软盘驱动器号(0=A)或硬盘驱动器号(80H=C)
ES:BX	数据区内 I/O 缓冲区的地址, 数据区应足够大以容纳所有要读取的扇区。 (BX 和 ES 配合使用)

下面的例子是读一个扇区到 SECTOR 的区域:

```

SECTOR    DB      512    DUP(?)      ; 输入区
...
MOV       AH, 02H        ; 请求读扇区
MOV       AL, 01         ; 一个扇区
LEA       BX, SECTOR     ; 输入缓冲区 (ES:BX)
MOV       CH, 05         ; 磁道 05
MOV       CL, 03         ; 扇区 03
MOV       DH, 00         ; 读写头 00
MOV       DL, 00         ; 0=驱动器 A, 80H=C, 等等
INT       13H            ; 调用中断服务

```

有效操作清除 CF, 并在 AL 中返回操作实际读取的扇区数。保持 DS、BX、CX 和 DX 的内容。操作出错则设置 CF, 并且 AH 中返回状态码, 重置驱动器(功能 00H), 然后重试这个操作。

大多数情况下, 只指定的一个扇区或者一个磁道上的全部扇区。初始化 CH 和 CL, 然后加 1 顺序读取下一个扇区。一旦扇区号超过了磁道最大扇区数时, 扇区号重置为 01, 并且在磁盘同一盘面上的磁道号加 1, 或者对下一盘面的读写头号加 1。

测试软盘是否准备好。程序可能发出请求访问一个尚未插入的软盘。标准操作是尝试操作 3 次后显示信息给用户。下例是试图使用 INT 13H 的功能 02H 读一个扇区的数据。试着使

用 DEBUG 输入指令(但没有语句号或注释), 在驱动器 A 中有磁盘或无磁盘的情况下测试程序代码。对于已装入软盘的情况, 操作应该读出磁盘根目录下的内容, 根目录有 512(200H) 字节, 起始位置为 DS:200H。程序代码如下:

```

0100    MOV     CX, 03           ; 循环计数
0103    PUSH   CX              ; 保存计数
0104    MOV     AX, 0201        ; 请求读一个扇区
0107    MOV     BX, 0200        ; 输入地址
010A    MOV     CX, 0001        ; 磁道和扇区号
010D    MOV     DX, 000C        ; 读写头和驱动器号
0110    INT     13H            ; 调用中断服务
0112    POP     CX              ; 恢复计数
0113    JNC     118             ; 如果没错, 退出
0115    CLC                    ; 如果出错
0116    LOOP    103             ; 尝试 3 次
0118    JMP     100             ; 再次执行

```

19.3.4 INT 13H 的功能 03H: 写扇区

与功能 02H 相反, 此操作从内存写一个特定区域(512 字节或 512 的倍数)到已格式化的指定扇区。加载寄存器和文件代号的过程同功能 02H。一个有效操作清零 CF, 并传递写入的扇区数到 AL, 保持 DS, BX, CX, DX 中的内容。操作出错, 设置 CF, 在 AH 中返回状态码, 重置驱动器并重试操作。

19.3.5 程序: 使用 INT 13H 读扇区

图 19-2 程序中使用 INT 13H 从磁盘读数据到存储器。注意, 没有打开操作或文件代号。主要数据区如下:

- BEGINADR 包含起始磁道(03)和扇区(01), 它们在程序中增量。
- ENDADR 包含结束磁道(04)和扇区(01)。扇区的总数是 9(磁道 3 的)×2(2 个盘面)=18。
改进程序的一种方法是给用户提示开始的和结束的磁道和扇区。
- SECTOR 为要读取的扇区定义一个 512 字节的数据区。

主要过程如下:

- A10MAIN 调用 B10ADDR、C10READ 和 D10DISPLY, 读入最后一个请求的扇区后结束。
- B10ADDR 根据盘面、磁道和扇区计算每个磁盘地址。当扇区数到达 19 的时候, 程序重新设置扇区为 01。如果盘面是 1, 程序增加磁道数, 然后改变盘面号, 从 0 到 1 或者从 1 到 0。这个过程只处理每磁道 18 个扇区的软盘(因为软盘是双面)。
- C10READ 从驱动器 A 读一个扇区到 SECTORIN, 有效读操作后扇区号加 1。
- D10DISPLY 显示当前读取的扇区的内容(INT 10H 的功能 13H 对回车和换行起作用), 并等待用户按任意键继续。

在 DEBUG 下运行程序, 从初始化段寄存器的指令开始跟踪。对于输入操作, 判定开始

和结束扇区在 FAT 中的位置(参见第 16 章)。用 G(go)命令执行程序,检测 FAT 和 SECTORIN 的目录入口项。

程序也能将输入区的 ASCII 字符转换成对等的十六进制数并且显示出来,正如 DEBUG 所做的一样(见图 14-5 程序)。用这种方法可以检验任意扇区的内容——甚至隐藏的内容,允许用户改变输入数据并将改变后的扇区内容写回磁盘。

```

TITLE      A19BIORD (EXE) 利用 BIOS 读磁盘扇区
.MODEL     SMALL
.STACK     64
.DATA

BEGINADR   DW      0301H           ;起始磁道-扇区
ENDADR     DW      0401H           ;结束磁道-扇区
ENDCODE    DB      00             ;结束处理标志
READMSG    DB      '*** Read error ***'
SECTORIN   DB      512 DUP(' ')   ;扇区输入区
SIDE       DB      00

; -----
.CODE
A10MAIN    PROC     FAR
MOV        AX,@data           ;初始化
MOV        DS,AX              ; 段寄存器
MOV        ES,AX              ;

A20:
MOV        AX,0003H           ;设置显示方式
INT        10H                ; 并清屏
CALL       B10ADDR            ;计算磁道地址
MOV        CX,BEGINADR        ;开始、结束
MOV        DX,ENDADR          ; 地址
CMP        CX,DX              ;在结束扇区?
JE         A90                ; 是,退出
CALL       C10READ            ;读磁盘记录
CMP        ENDCODE,00         ;正常读?
JNZ        A90                ; 否,退出
LEA        BP,SECTORIN        ;扇区地址和
MOV        CX,512              ; 长度
CALL       D10DISPLY          ;显示扇区
JMP        A20                ;重复操作

A90:
MOV        AX,4C00H           ;结束处理
INT        21H

A10MAIN    ENDP
;
; 计算下一个磁道/扇区:
B10ADDR    PROC     NEAR
MOV        CX,BEGINADR        ;使用 CX 寄存器
CMP        CL,19              ;得到磁道/扇区
JNE        B90                ;通过最后一个扇区(19)?
MOV        CL,01              ; 否,退出
CMP        SIDE,00            ; 是,设置扇区为 1
JE         B20                ;如为盘面 0, 跳转
INC        CH                  ;磁道加 1
XOR        SIDE,01            ;改变盘面
MOV        BEGINADR,CX
B20:
KOR
MOV
B90:
RET
B10ADDR    ENDP
;
; 读磁盘扇区内容:
C10READ    PROC     NEAR
MOV        AH,02H             ;使用 AX, BP, BX, CX, DX
MOV        AL,01              ;请求读
MOV        BX,SECTORIN        ;扇区数
LEA        BX,SECTORIN        ;缓冲区地址
MOV        CX,BEGINADR        ;磁道/扇区

```

图 19-2 使用 INT 13H 读磁盘扇区

```

        MOV     DH, SIDE           ; 盘面
        MOV     DL, 00            ; 驱动器 A
        INT     13H
        JNC     C90               ; 如正常读, 退出
        MOV     ENDCODE, 01       ; 否则,
        LEA     BP, READMSG       ; 显示错误信息
        MOV     CX, 18            ;
        CALL    D10DISPLY         ;
C90:    INC     BEGINADR          ; 扇区加 1
        RET
C10READ ENDP
;
; ----- 显示程序: ----- ; 使用 AX, BX, DX
D10DISPLY PROC NEAR              ; BP, CX 设置
        MOV     AX, 1301H        ; 请求显示
        MOV     BX, 0016H        ; 页和属性
        MOV     DX, 0300H        ; 行: 列
        INT     10H
        MOV     AH, 10H          ; 等待键盘
        INT     16H              ; 输入
        RET
D10DISPLY ENDP
END      A10MAIN

```

图 19-2 续

当用 INT 21H 写文件时, 在可用的簇中插入记录, 这些簇在磁盘中可能不相邻。由于这个原因, 不能期望 INT 13H 顺序地读出这个文件, 但可以访问 FAT 的入口项来得到下一簇的位置。

19.4 其他 INT 13H 磁盘操作

以下部分描述附加的 INT 13H 磁盘服务例程。

19.4.1 INT 13H 的功能 04H: 检测扇区

这一操作简单检查指定的可读扇区, 并且执行柱面冗余检验(CRC)。当一个操作写数据到扇区时, 磁盘控制器根据设置的位计算 CRC 校验和, 并写入紧接着的扇区。可以使用功能 04H 读扇区, 重新计算校验和, 并与存储的值相比较。核对数只由重新计算的校验和组成, 它不检验扇区内的字节值是否和内存中数据一致。在写(功能 03H)之后可用此功能来确保更可靠的输出, 但要花费更多的处理时间。

加载寄存器和功能 02H 相同, 但是由于操作并不检验所写数据, 所以不需要设置 ES:BX 地址。一个成功的操作清除 CF, 并在 AL 中返回实际检测的扇区数, 保持 DS, BX, CX, DX 中的内容。操作出错设置 CF, 在 AH 中返回状态码, 并且重置驱动器, 重试此操作。

19.4.2 INT 13H 的功能 05H: 格式化轨迹

读/写操作要求磁道的格式化信息来定位和访问请求的扇区。这个操作根据 4 种不同的大小来格式化磁道。在执行此操作之前, 要使用功能 17H 设置软盘类型, 使用功能 18H 设置媒

体类型。为了格式化磁盘，初始化寄存器如下：

AL	要格式化的扇区数
CH	柱面/磁道号(从 0 开始)
DH	读写头/盘面号(软盘是 0 或 1)
DL	软盘驱动器号(0=A)或硬盘驱动器号(80H=C)
ES:BX	段:偏移地址指向磁道的一组地址字段。对磁道上的每个扇区都有一个 4 字节的入口项，格式为 T/H/S/B： 字节 0 T=柱面/磁道号 1 H=读写头/盘面号 2 S=扇区号 3 B=每扇区的字节数(00H=128, 01H=256, 02H= 512, 03H=1024)

例如，要格式化磁道 03，读写头 00，每个扇区 512 字节，磁道的第一个入口项是 03000102H，接着是每个剩余扇区的一个入口项。

该操作清除(如有效)或设置(如无效)进位标志，并在 AH 中返回状态码。

19.4.3 INT 13H 的功能 08H：获取驱动器参数

这个很有用的功能返回有关磁盘驱动器的信息。在 DL 中装入驱动器号(软盘：0=A, 1=B, 硬盘：80H 或更大)。成功操作返回如下信息：

BL	软盘类型(01H=360K, 02H=1.2M, 03H= 720K, 04H=1.44M)
CH	柱面/磁道号的高位
CL	位 7~6=柱面号的高 2 位 位 5~0=扇区号的高位
DH	读写头号的高位
DL	与控制器连接的驱动器数
ES:DI	对于软盘，是 11 字节的软盘驱动器参数表的段:偏移地址。两个相关的字段是： 偏移地址 3 给出每个扇区的字节数(00H=128, 01H=256, 02H= 512, 03H=1024) 偏移地址 4 给出每个磁道的扇区数。

操作清除(如有效)或者设置(如无效)进位标志，并在 AH 中返回状态码。试使用 DEBUG 命令 D ES:偏移地址(返回在 DI 中的偏移地址)来显示驱动器参数。

19.4.4 INT 13H 的功能 09H：初始化驱动盘

当根据计算机自带的硬盘表启动计算机时，BIOS 执行此功能。DL 中为驱动器号(80H 或更大)。操作清除(有效)或者设置(无效)进位标志，并在 AH 中返回状态码。相关操作有 BIOS INT 41H 和 INT 46H。

19.4.5 INT 13H 的功能 0AH: 读扩展扇区缓冲区

硬盘上的扇区缓冲区包括 512 字节的数据再加上 4 个字节的纠错码(ECC)，用来检查错误并纠正。这一功能可以读整个扇区缓冲区，不仅仅是数据区部分。为了读扩展缓冲区，加载如下寄存器：

AL	扇区数（最多为一个驱动器的最大扇区数）
ES:BX	段:偏移量，输入缓冲区的地址
CH	柱面/磁道号
CL	位 7~6=柱面号的高 2 位 位 5~0=扇区号高位
DH	读写头/盘面号
DL	驱动器号(80H 或更高)

成功操作在 AL 中返回传输的扇区数。操作清除(有效)或者设置(无效)进位标志，并在 AH 中返回状态码。

19.4.6 INT 13H 的功能 0BH: 写扩展扇区缓冲区

这一功能和功能 0AH 相似，但这一功能是把缓冲区的内容(包括 ECC 码)写到硬盘上。

19.4.7 INT 13H 的功能 0CH: 搜索磁盘柱面

这一功能可以把硬盘的读写头定位到某一指定的柱面(磁道)上，但不传送任何数据。为了搜索柱面，加载如下寄存器：

CH	柱面/磁道号
CL	位 7~6=柱面/磁道号的高 2 位 位 5~0=扇区号
DH	磁头/盘面号
DL	驱动器号(80H=C)

操作清除(有效)或者设置(无效)CF 并且在 AH 中返回状态码。

19.4.8 INT 13H 的功能 0DH: 交替磁盘重置

除了只适用于硬盘外，这一操作和功能 00H 相同。在 DL 中设置驱动器号(80H 或更大的值)。操作重置读写访问臂到柱面 0。操作清除(有效)或者设置(无效)CF，并在 AH 中返回状态码。

19.4.9 INT 13H 的功能 0EH: 读扇区缓冲区

这一操作和功能 0AH 一样, 它只读取 512 字节的数据区, 不读取 ECC 字节。

19.4.10 INT 13H 的功能 0FH: 写扇区缓冲区

这一操作和功能 0BH 一样, 它只写 512 字节的数据区, 不写 ECC 字节。

19.4.11 INT 13H 的功能 10H: 测试驱动器是否准备好; 11H: 重新校准硬盘驱动器; 12H: 诊断 ROM; 13H: 诊断驱动器; 14H: 诊断控制器

这些功能执行内部诊断, 并将特殊信息报告给 BIOS 和高级应用程序。操作清除(有效)或者设置(无效)CF 并且在 AH 中返回状态码。

19.4.12 INT 13H 的功能 15H: 得到磁盘类型

这一功能返回磁盘驱动器的信息。对于磁盘, 在 DL 中设置驱动盘号(0=A, ...); 对于硬盘, 在 DL 中设置 80H 或更大的值(80H=C, 81H=D, ...)。有效操作在 AH 中返回如下代码之一:

00H	当前没有驱动器/磁盘
01H	软盘驱动器不检测换盘
02H	软盘驱动器检测换盘
03H	软盘驱动器

当 AH 中返回 03H 时, CX:DX 包含了驱动器上硬盘扇区的总数。操作清除(有效)或设置(无效)进位标志并且在 AH 中返回状态码。

19.4.13 INT 13H 的功能 16H: 改变磁盘状态

这一功能为能够检测换盘的系统检测换磁(参见功能 15H)。在 DL 中设置驱动盘号(0=A, ...)。操作在 AH 中返回如下代码之一:

00H	没有换磁(CF=0)
01H	无效磁盘参数(CF=1)
06H	换盘(CF=1)
80H	软盘驱动器未准备好(CF=1)

状态 01H 和 80H 是错误状态, 所以要设置 CF, 而 06H 是一种有效状态但也设置 CF——这是容易造成混淆的一个原因。

19.4.14 INT 13H 的功能 17H: 设置软盘类型

这一操作设置软盘和驱动器的组合。一起使用功能 17H 和功能 05H 可以格式化磁盘。在 DL 中设置驱动器号(0=A, ...), 在 AL 中加载软盘类型。软盘类型包括: 01H=360K 软盘及 360K 驱动器, 02H=360K 软盘及 1.2M 驱动器, 03H=1.2M 软盘及 1.2M 驱动器, 04H=720K 软盘及 720K 驱动器, 05H=1.44M 软盘在 1.44M 驱动器。操作清除(有效)或者设置(无效)CF 并且在 AH 中返回状态码。

19.4.15 INT 13H 的功能 18H: 设置格式化媒体类型

此操作紧挨着功能 05H 之前执行。为了设置媒体类型, 加载如下寄存器:

CH	磁道数(低 8 位)
CL	磁道数(高 2 位在位 7~6), 每个磁道的扇区数(位 5~0)
DL	驱动器(0=A, ...)

有效操作在 ES:DI 中返回一个指向 11 字节的磁盘参数表(参见功能 08H)。操作清除(有效)或者设置(无效)CF 并且在 AH 中返回状态码。

19.4.16 INT 13H 的功能 19H: 停置磁头

较早的磁盘驱动器在关掉系统后, 将读写头置于驱动器表面上, 所以必须将读写头停置到一个安全区域。目前的大部分驱动器都可以自动停置磁头。操作要求在 DL 中加载驱动器号(对于硬盘, 80H 或更大)。操作清除(有效)或者设置(无效)CF, 并在 AH 中返回状态码。

19.5 要 点

- BIOS INT 13H 提供了对磁道和扇区的直接访问。操作包括重置/读磁盘状态, 读/写扇区和格式化磁道。
- INT 13H 不能提供自动目录处理, 文件结束操作, 或记录的分块和解决功能。
- 检验扇区操作对写数据进行基本检验, 这要花费一定的执行时间。
- 程序在每个 INT 13H 磁盘操作后要检验状态字节。

19.6 习 题

- 19-1. 使用 BIOS INT 13H 的两个主要缺点是什么? 为什么经常使用 INT 21H?
- 19-2. 在什么情况下程序员可以使用 INT 13H?

19-3. INT 13H 操作返回一个状态码。(a)返回的状态码在哪里? (b)代码 00H 的含义是什么? (c)代码 03H 的含义是什么?

19-4. INT 13H 返回错误的标准过程是什么? 即如何检验出错, 如何处理?

19-5. 编写重置磁盘控制器的程序。

19-6. 编写读磁盘状态的程序。

19-7. 使用驱动器 A, 读写头 0, 磁道 4, 扇区 6, 用 INT 13H 指令编写读 4 个扇区到数据区 SECTORS 的程序。

19-8. 使用存储器地址 SECTOR, 驱动器 A, 读写头 0, 磁道 6, 扇区 4, 用 INT 13H 指令编写程序写一个扇区。注意使用备用磁盘来做这个练习。

19-9. 在上题中, 写操作后, 怎样检验是写一个写保护的磁盘?

19-10. 在 19-8 基础上, 编写检验写操作的程序。

目的：描述使用各种中断操作进行打印的必要条件。

20.1 引言

与屏幕和磁盘处理相比，打印处理显得相对简单，它只包括几个操作，通过 INT 21H 或者 BIOS INT 17H 指令完成。发送给打印机的特殊命令包括：换页、换行、Tab 和回车。

打印机必须理解发自处理器的信号，例如，走一页纸，向下换一行，或者换页定位。处理器也必须理解表示打印机“正忙”或者“缺纸”的信号。不幸的是，不同类型的打印机对于处理器信号的回应是不同的，而软件专业人员的任务之一就是使程序与这些打印机配接。

本章介绍了用于处理打印机的中断操作如下：

INT 21H 功能

40H 打印字符

INT 17H 功能

00H 打印字符

01H 初始化端口

02H 获取打印机端口状态

20.2 普通打印机控制符

用于控制 PC 机上使用的普通打印机的标准字符如下：

十进制	十六进制	功能
09	09H	水平 Tab
10	0AH	换行(前进一行)
12	0CH	换页(前进到下一页)
13	0DH	回车(回到左页边)

1. 水平 Tab

水平 Tab(09H)控制符使打印机从当前的打印位置前进到下一个 Tab 停止处(如果已设置,通常每 8 个位置即停)。该命令只对有此功能的打印机,并且打印机的 Tab 功能已被设置时有效。如果打印机没有打印 Tab 的能力,可以使用一连串的空格字符来代替。

2. 换行

换行(OAH)控制符使打印机前进一行，两个连续的换行符可以得到两个空行。

3. 换页

当启动打印机时,初始的页决定了每页顶端的开始位置。一页默认的长度是 11 英寸,按照每英寸 6 行计算,每页有 66 行。处理器和打印机都不会自动检查页的底端。无论在一台激光打印机上使用分页格式还是连续打印格式,程序员要负责指示打印机开始下一页的打印。为了控制换页,可以在打印的同时计算行数,当达到一页的最大行数(比如 60 行)时,使用换页(OCH)控制命令,然后将程序的行计数重新设置为 0 或 1。

打印结束时，传送换行或者换页控制命令强制打印机将仍在缓冲区中的最后一行打印出来。在打印结束发出的换页符可以保证最后一页送出打印机。

4. 回车

回车(ODH)控制字符通常和换行命令一起使用,可将打印机重新置于它最左面的页边。该字符即为键盘上的<Enter>键或<Return>键,屏幕显示时为CR。

20.3 INT 21H 的功能 40H: 打印字符

在关于屏幕处理和磁盘处理的章节中，我们已经用到了文件代号。为了使用 INT 21H 的功能 40H 打印，需设置下列寄存器：

AH=功能 40H CX=打印的字符数

BX=文件代号 04 DX=打印数据的地址

下面的例子从左页边开始打印 **HEADING** 数据项中的 27 个字符。紧接着 **HEADING** 正文的回车符(ODH)和换行符(OAH)使打印机重置到 0 列并前进一行:

```
CR EQU 0DH ; 回车
LF EQU 0AH ; 换行
HEADING DB 'Mountain Outfitting Corp.', CR, LF
;
MOV AH, 40H ; 请求打印
MOV BX, 04 ; 打印机的文件代号 04
MOV CX, 27 ; 发送 27 个字符
LEA DX, HEADING ; 打印数据的地址
INT 21H ; 请求中断服务
```

成功的操作将打印正文，清除进位标志，并且在 AX 中返回打印的字符数。不成功的操作会

设置进位标志，并在 AX 中返回错误代码 05(拒绝存取)或者 06(非法文件代号)。传输数据中的文件尾标记(Ctrl-Z 或者 0AH)也会终止操作。

下面两种情况会中断试图打印的动作：

1. 打印机电源没有打开。系统显示

“Write Fault Error Writing Device PRN”

“Abort Retry Ignore Fail”

2. 缺纸或卡纸。系统显示

“Printer out of paper error writing device PRN”

20.3.1 程序：具有页溢出处理和标题的打印程序

前面图 8-2 中的程序从键盘接受名字并在屏幕上显示出来。图 20-1 中的程序与此相似，不同的是将名字发送给打印机。打印出来的每一页都包括一个标题，紧接着两个空格，输入姓名的打印格式如下：

```
List of Customer Names      Page 01
Annie Hall
Fanny Hill
Danny Rose
--
```

该程序对每个打印行计数，当接近一页的底端时，将格式换到下一页的顶端。主要的过程如下：

- A10MAIN 调用 B10INPUT 和 C10PRINT，当用户仅按下回车键时结束处理。
- B10INPUT 提示输入并且接受从键盘输入的姓名。
- C10PRINT 调用 D10PAGE(如果在页尾)并且打印姓名(其长度取决于键盘输入参数列表中的实际长度)。
- D10PAGE 前进到新的一页，打印标题，重置行计数，并使页计数增加。
- P10OUT 处理打印请求。

```
TITLE      A20PRTNM (EXE) 接收输入的姓名并打印
.MODEL     SMALL
.STACK    64
.DATA
NAMEPAR    LABEL    BYTE           ; 键盘参数表:
MAXNLEN    DB        20           ; 名字最大长度
NAMELEN    DB        ?            ; 实际输入的长度
NAMEFLD    DB        20 DUP(' ') ; 输入的名字
HEAD_LEN   EQU       37           ; 标题行:
HEADING     DB        'List of Customer Names      Page
PAGECTR     DB        '01', 0DH, 0AH, 0AH

BOTTPAGE    EQU       60
FORMFEED    DB        0CH          ; 换页
LINEFEED    DB        0DH, 0AH     ; 回车, 换行
LINECTR     DB        01           ; 打印行计数
```

图 20-1 具有页溢出处理和标题的打印程序


```

PROMPT DB 'Name? '
ROW DB 00 ; 屏幕行
.386 ; -----
; CODE
A10MAIN PROC FAR
MOV AX,@data ; 初始化
MOV DS,AX ; 段寄存器
MOV ES,AX ;
MOV AX,0003H ; 设置显示方式
INT 10H ; 并清屏
CALL D10PAGE ; 页标题

A20: CALL B10INPUT ; 接收输入名
CMP NAMELEN,00 ; 名字输入?
JE A30 ; 否, 退出
CALL C10PRINT ; 是, 准备打印
JMP A20

A30: MOV CX,01 ; 处理结束:
LEA DX,FORMFEED ; 换页符
CALL P10OUT ;
MOV AX,4C00H ; 退出
INT 21H

A10MAIN ENDP
;
; 从键盘接收输入的名字:
; -----
B10INPUT PROC NEAR ; 使用 AX, BP, BX, CX, DX
MOV AX,1301H ; 显示提示符
MOV BX,0016H ; 页和属性
LEA BP,PROMPT ; 提示符地址
MOV CX,06 ; 行长度
MOV DH,ROW ; 行
MOV DL,10 ; 列
INT 10H
INC ROW ; 设置下一行
MOV AH,0AH ; 请求键盘
LEA DX,NAMEPAR ; 输入
INT 21H
RET
B10INPUT ENDP
;
; 准备打印名字:
; -----
C10PRINT PROC NEAR
CMP LINECTR,BOTTPAGE ; 页尾?
JB C20 ; 否, 跳过
CALL D10PAGE ; 打印标题

C20: MOVZX CX,NAMELEN ; 设置字符数
LEA DX,NAMEFLD ; 设置名字地址
CALL P10OUT ; 打印名字
MOV CX,02 ; 请求回车
LEA DX,LINEFEED ; 换行
CALL P10OUT ;
INC LINECTR ; 行计数加1
RET
C10PRINT ENDP
;
; 打印标题和页号:
; -----
D10PAGE PROC NEAR ; 使用 CX, DX
CMP WORD PTR PAGECTR,3130H ; 第1页?
JE D30 ; 是, 跳过
MOV CX,01 ;
LEA DX,FORMFEED ; 否, 执行
CALL P10OUT ; 换页
MOV LINECTR,03 ; 重置行计数

D30: MOV CX,HEAD_LEN ; 标题长度,
LEA DX,HEADING ; 标题地址
CALL P10OUT
INC PAGECTR+1 ; 页计数加1
CMP PAGECTR+1,3AH ; 个位 = 10?
JNE D90 ; 否, 跳过
MOV PAGECTR+1,30H ; 是, 个位为0
INC PAGECTR ; 十位加1

```

图 20-1 续

```

D90:      RET
D10PAGE  ENDP
;
;          ----- 打印程序: -----
;          ; 使用 AH, BX
;          ; CX, DX 已设置
P10OUT    PROC    NEAR
MOV       AH, 40H    ; 请求打印
MOV       BX, 04     ; 文件代号
INT       21H
RET
P10OUT    ENDP
END       A10MAIN

```

图 20-1 续

在执行的开始, 程序必须打印标题, 但是不走纸到新的一页。结束时, 如果 PAGECTR 为其初始值 01, D10PAGE 跳过换页操作。PAGECTR 被定义为 DB '01', 这会产生一个 ASCII 数字 3031H。程序使 PAGECTR 增 1, 这样它就逐渐变为 3032, 3033, 以此类推。这个值一直增长到 3039 都是有效的, 此后将变为 303A, 这会打印出 0 和冒号。此时, 程序将个位数字重置, 3AH 变为 30H, 并且在十位数字上加 1, 所以, 303AH 变为 3130H, 或者十进制数字 10。PAGECTR 的容量是 99。

在每页的结尾打印姓名之前(而不是之后)设置一个检验, 以确定在最后一页的标题之下至少有一个名字。可以通过添加滚动来改进程序。

20.3.2 程序: 打印 ASCII 文件并处理 Tab

一个通用程序, 例如显示系统所执行的程序, 在到达下一个可以被 8 整除的位置之前用空白来代替 Tab 符(09H)。这样 Tab 的终止处可以在位置 8, 16, 24, 以此类推, 所以在 0 到 7 之间的位置都定位到 8, 而在 8 到 15 之间的定位到 16, 以此类推。但是, 有些打印机对 Tab 符不起作用。一个打印 ASCII 文件(例如汇编源文件)的程序, 应该检查每一个传送到打印机的字符。如果字符是 Tab, 程序应插入空白一直到下一个定位位置。

图 20-2 的程序要求用户键入一个文件名, 并打印该文件的内容。这个程序与图 17-3 中显示记录的程序相似, 更进一步的处理是, 用一段空白代替打印机的 Tab 终止位置。下面是 3 个 Tab 终止位置的例子, 其打印位置是 1、9 和 21, 设置下一个 Tab 位置的逻辑如下:

当前打印位置	1	9	21
二进制数值	00000001	00001001	00010101
清除最右面 3 位	00000000	00001000	00010000
加 8	00001000	00010000	00011000
新的 Tab 终止位置	8	16	24

程序的组织如下:

- A10MAIN 调用 B10PROMPT, C10OPEN, D10READ 和 E10XFER。
- B10PROMPT 要求用户键入文件名。按回车键表示结束。
- C10OPEN 打开要求的磁盘文件作为输入。如果操作是合法的, 程序使用 INT 21H 的功能 42H 来确定文件的大小(仅用低位部分, 最大为 65 535 字节)。
- D10READ 从文件读入一个扇区。
- E10XFER 检查输入的数据, 查找扇区尾, 文件尾, 显示区的结尾, 换行和 Tab。这

个过程基本上向打印区发送的都是规则的字符，并且用上述逻辑处理来定位 Tab 的终止位置。这个程序通过每个字符处理后存储的文件大小递减 1 来确定文件的末尾。

- P10PRINT 打印输出行，并将其清除为空白。

可以修改程序的打印行计数，并且在靠近一页的底端，在 60 行左右强制换页；也可以使用一个编辑程序直接将换页符插入到你的 ASCII 文件中，恰好在一页终止的位置上，就像在一个过程的结尾一样。通常的方法是按下 Alt 键和数字键盘上的数字键，例如换页是 012。

```

TITLE      A20PRTAS (EXE) 读取并打印 ASCII 记录
.MODEL     SMALL
.STACK    64
.DATA
NAMEPAR    LABEL    BYTE           ; 输入文件名参数表
MAXLEN     DB       32             ;
NAMELEN    DB       ?             ;
FILENAME    DB       32 DUP(' ')

FILEDTA    LABEL    BYTE           ; 为磁盘文件 DTA
DB          26 DUP(20H)           ; 保留的空
FILESIZE    DW       0             ; 文件大小 (低位)
DW          0                     ; 文件大小 (高位)
DB          13 DUP(20H)           ; 其余的文件 DTA
PRINT_LEN   EQU      120
COUNT      DW       00
ENDCODE     DW       00           ; 结束处理标志
FORMFEED    DB       0CH
HANDLE      DW       0
OPENMSG     DB       '*** Open error ***'
PRNTAREA    DB       120 DUP(' ') ; 打印区
PROMPT      DB       'Name of file? '
ROW         DB       0           ; 屏幕行
SECTOR      DB       512 DUP(' ') ; 文件输入区
.386 ;
-----
.CODE
A10MAIN     PROC     FAR           ; 主程序
MOV         AX,@data             ; 初始化段寄存器
MOV         DS,AX
MOV         ES,AX
MOV         AX,0003H             ; 设置显示方式
INT         10H                  ; 清屏

A20:        MOV         ENDCODE,00 ; 初始化
CALL        B10PROMPT            ; 请求文件名
CMP         NAMELEN,00           ; 有请求?
JE          A90                  ; 无, 退出
CALL        C10OPEN              ; 打开文件, 得到文件代号
CMP         ENDCODE,00           ; 有效打开?
JNE         A20                  ; 否, 重试
CALL        D10READ              ; 读磁盘第 1 扇区
CMP         ENDCODE,00           ; 文件尾, 无数据?
JE          A80                  ; 是, 请求下一个扇区
CALL        E10XFER              ; 打开/读

A80:        MOV         AH,3EH     ; 关闭文件
MOV         BX,HANDLE
INT         21H
JMP         A20                  ; 重复处理

A90:        MOV         AX,4C00H   ; 结束处理
INT         21H

A10MAIN     ENDP

;
; 请求用户输入文件名:
;
B10PROMPT   PROC     NEAR         ; 使用 AH, BP, BX, CX, DX
LEA         BP,PROMPT            ; 提示符地址
MOV         CX,13                ; 行长度

```

图 20-2 打印 ASCII 文件

```

        CALL F10DISPLY
        MOV AH, 0AH                ; 从键盘接收
        LEA DX, NAMEPAR            ; 文件名
        INT 21H
        MOVZX BX, NAMELEN          ; 在文件说明
        MOV FILENAME[BX], 00H     ; 末尾加入 0
        RET
B10PROMPT ENDP
;
; 打开磁盘文件:
; -----
C10OPEN PROC NEAR                ; 使用 AX, BP, CX, DX
        MOV AX, 3D00H            ; 打开, 只读
        LEA DX, FILENAME
        INT 21H
        JNC C20                  ; 检测 CF
        LEA BP, OPENMSG          ; CF = 1, 出错
        MOV CX, 18               ; 显示错误信息并结束
        CALL F10DISPLY
        MOV ENDCODE, 01
        JMP C90
C20:    MOV HANDLE, AX           ; 保存文件代号
        MOV AH, 1AH             ; 设置 DTA
        LEA DX, FILEDTA
        INT 21H
        MOV AH, 4EH             ; 找到文件并得到文件大小
        MOV CX, 0
        LEA DX, FILENAME
        INT 21H
C90:    RET
C10OPEN ENDP
;
; 读磁盘扇区
; -----
D10READ PROC NEAR                ; 使用 AH, BX, CX, DX
        MOV AH, 3FH             ; 请求读
        MOV BX, HANDLE          ; 设备
        MOV CX, 512             ; 长度
        LEA DX, SECTOR          ; 缓冲区
        INT 21H
        MOV ENDCODE, AX
        RET
D10READ ENDP
;
; 传输数据到打印行:
; -----
E10XFER PROC NEAR                ; 使用 AL, BX, DI, DX, SI
        CLD                     ; 从左到右
        LEA SI, SECTOR          ; 初始化
E20:    LEA DI, PRNTAREA
        MOV COUNT, 00
E30:    LEA DX, SECTOR+512
        CMP SI, DX              ; 扇区尾?
        JNE E40                ; 否, 跳过
        CALL D10READ            ; 是, 读下一个扇区
        CMP ENDCODE, 00         ; 文件尾?
        JE E30                  ; 是, 退出
        LEA SI, SECTOR
E40:    MOV BX, COUNT
        CMP BX, PRINT_LEN       ; 在打印区末尾?
        JB E50                  ; 否, 跳过
        MOV [DI+BX], 0D0AH      ; 是, 设置回车/换行
        CALL P10PRINT
        LEA DI, PRNTAREA
        MOV COUNT, 00           ; 重新初始化
E50:    LODSB
        MOV BX, COUNT           ; [SI] 放入 AL, SI 加 1
        MOV [DI+BX], AL        ; 字符送到打印行
        INC BX
        DEC FILESIZE
        JZ E90                  ; 所有字符均处理?
        CMP AL, 0AH            ; 是, 退出
        JNE E60                ; 换行?
        JNE E60                ; 否, 跳过

```

图 20-2 续

```
CALL P10PRINT          ; 是, 调用打印程序
JMP E20
E60:  CMP AL,09H          ; Tab字符?
      JNE E70            ; 否, 跳过
      DEC BX             ; 是, 重置 BX
      MOV BYTE PTR [DI+BX],20H ; 清除tab为空
      AND BX,0FFF8H      ; 清零最右 3 位
      ADD BX,08          ; 加 8 为tab的终止位置
E70:  MOV COUNT,BX
      JMP E30
E90:  MOV BX,COUNT        ; 文件尾
      MOV BYTE PTR [DI+BX],0CH ; 换页
      CALL P10PRINT      ; 打印最后一行
      RET
E10XPER ENDP
;
; Display routine:
; -----
F10DISPLY PROC NEAR      ;使用 AX, BX, DX
; BP, CX 已设置
      MOV AX,1301H      ;请求显示
      MOV BX,0016H      ;页:属性
      MOV DH,ROW        ;行
      MOV DL,10         ;列
      INT 10H
      INC ROW           ;下一个屏幕行
      RET
F10DISPLY ENDP
;
; 打印行并清除为空:
; -----
P10PRINT PROC NEAR      ;使用 AX, BX, CX, DI, DX
      MOV AH,40H        ;请求打印
      MOV BX,04
      MOV CX,COUNT      ;行长度
      INC CX
      LEA DX,PRNTAREA
      INT 21H
      MOV AL,20H        ;清除打印行
      MOV CX,PRINT_LEN
      LEA DI,PRNTAREA
      REP STOSB
      RET
P10PRINT ENDP
      END A10MAIN
```

图 20-2 续

20.4 专用打印机控制符

前面的部分描述了基本的打印机控制符: Tab、换行、换页和回车的用法。其他适用于大多数打印机的命令如下:

十进制	十六进制	作用
08	08	退格
11	0B	垂直 Tab
14	0E	打开扩展模式
15	0F	打开紧缩模式
18	12	关闭紧缩模式
20	14	关闭扩展模式

一些打印命令需要一个 Esc 符(1BH)在前面:

1B 30 设置行间距为每英寸 8 行

1B 32 设置行间距为每英寸 6 行

1B 45 开始加重打印模式

1B 46 结束加重打印模式

可以用两种方式给打印机发送控制符：

1. 在数据区定义控制字符。下面的语句设置了紧缩模式，设置每英寸 8 行，打印标题，并使用了一个回车和换行：

```
HEADING DB 0FH, 1BH, 3CH, 'Mountain Outfitting Corp.', 0DH, 0AH
```

2. 使用功能 40H 发送控制字符给打印机：

```
CONDMODE    DB 0FH                ; 紧缩模式
MOV AH, 40H  ; 请求打印
MOV BX, 04   ; 文件代号
MOV CX, 01   ; 字符数
LEA DX, CONDMODE ; 紧缩模式
INT 21H      ; 调用中断服务
```

接下来的所有字符都以紧缩模式打印，直到程序发送另一条命令重置打印模式。

20.5 BIOS INT 17H 打印功能

INT 17H 提供了在 BIOS 级上打印的功能。合法的打印机端口 LPT1、LPT2、LPT3 分别是 0 (默认)、1 和 2。INT 17H 提供的 3 种功能，在 AH 寄存器中指定：

1. 先调用功能 02H，通过一个选定的端口号来确定打印机的状态。每次试图打印之前进行该状态的测试。如果打印机可用，则执行 2.。

2. 调用功能 01H 来初始化打印机端口。

3. 调用功能 00H 操作给打印机发送字符。

该操作返回打印机的状态到 AH，将一位或更多的位置为 1：

位	状态	位	状态
0	超时	5	缺纸
3	输入/输出错误	6	得到打印机应答
4	选定	7	空闲

如果打印机已经打开，并准备好打印，该操作返回 90H (二进制 10010000)：打印机空闲，并且已被选定，这是一种合法的状态。表示打印机出错是第 5 位 (缺纸) 和第 3 位 (输出错) 为 1。如果打印机没有打开，该操作返回 B0H，或为二进制数 10110000，表明“缺纸”。

1. INT 17H 的功能 00H：打印一个字符。这个操作能打印一个字符，并允许请求打印机端口 0、1 或 2。打印的字符存放于 AL 中，打印机端口号存放于 DX 中：

```
MOV AH, 00H    ; 请求打印
MOV AL, char    ; 要打印的字符
MOV DX, 00     ; 选择打印机端口 0
INT 17H        ; 调用中断服务
```

该操作在 AH 中返回状态码。在实际操作时推荐先使用功能 02H 检验打印机状态。

2. INT 17H 的功能 01H: 初始化打印机端口。该操作选定一个打印机端口, 重置打印机, 并将其初始化, 以备接受数据。下例选择端口 0:

```
MOV     AH, 01H      ; 请求初始化端口
MOV     DX, 00       ; 选择打印机端口 0
INT     17H          ; 调用中断服务
```

因为该操作发送一个换页符给打印机, 可以用它设置格式到页顶的位置, 尽管大多数的打印机在启动时自动设置到页顶位置。该操作在 AH 中返回状态码。

3. INT 17H 的功能 02H: 获得打印机端口状态。该操作的目的是确定打印机的状态。下面以端口 0 为例:

```
MOV     AH, 02H      ; 请求读端口
MOV     DX, 00       ; 选择打印机端口 0
INT     17H          ; 调用中断服务
TEST    AH, 00101001B ; 准备就绪?
JNZ     errormsg      ; 否, 显示信息
```

该操作返回与功能 01H 一样的打印机端口状态。当程序运行时, 如果打印机开始没有打开, BIOS 不能自动返回信息——程序设计的是需要测试打印机状态, 并根据状态进行动作。如果程序不测试状态, 唯一的提示就是闪烁光标。如果在这时打开打印机, 一些输出数据将会丢失。所以, 在进行任何 BIOS 打印操作前, 要检验端口状态; 如果有错误, 显示信息 (INT 21H 自动执行这个检测, 虽然对各种错误返回的信息都是“缺纸”)。当打印机开关接通时, 信息不再显示, 并且开始正常打印, 而不会丢失数据。

在任何情况下, 打印机都可能超出格式或者被不小心关掉电源。如果你写的程序也供别人使用, 在每次试图打印之前都应该包括状态测试(功能 02H)。

20.6 要 点

- 标准的打印机控制符是水平 Tab、回车、换行和换页。
- 打印完成之后, 使用换行或换页命令清除打印机缓冲区。
- INT 21H 的功能 40H 打印字符串, 而 BIOS INT 17H 一次只能打印一个字符。
- 如果有打印机错误发生, 系统将显示信息, 虽然 BIOS 只返回一个状态码。当使用 BIOS INT 17H 时, 在打印之前用功能 02H 检测打印机状态。

20.7 习 题

20-1. 提供打印机控制字符: (a)回车, (b)换行, (c)换页, (d)水平 Tab。

20-2. 针对下列要求, 使用 INT 21H 的功能 40H 编写程序代码: (a)走纸到下页, (b)定义并且打印你的名字, (c)完成一次回车和换行, 并且打印你的街道地址, (d)完成一次回车和换

行，并且打印你的城市和州，(e)走纸。

20-3. 修改题 20-2 的程序使名字在扩展模式下打印，街道和地址在紧缩模式下打印，城市和州在加重模式下按正常大小打印。

20-4. 修改题 20-3 的程序，使其完成(b)、(c)、(d)5 次。

20-5. 修改图 20-3 的程序，使程序能显示打印出的行。

20-6. 定义一个有回车和换页操作的标题行，设置紧缩模式，定义标题(任意名称)，然后关闭紧缩模式。

20-7. INT 17H 在 AH 中返回一个打印错误码。下面的代码是什么意思：(a)08H, (b)10H, (c)90H。

20-8. 使用 INT 17H 修改题 20-2，包括测试打印机状态。

第六部分

特殊的课题

目的：说明宏指令的定义与使用

21.1 引 言

对于每一条符号指令，汇编程序都会产生一条机器语言指令。另一方面，用高级语言(如 C 或 BASIC)编写的每个语句，编译程序都会产生许多机器语言指令。关于这一点，可以把高级语言看成是由一组宏语句所组成。

汇编程序便于定义宏。为宏定义一个惟一的名称，同时定义一组汇编语言指令，这些指令是宏产生的。然后，无论在哪里，当需要编写这一组指令时，就简单地编写宏的名称，而汇编程序会自动地产生所定义的指令。

宏对于以下目的是有用的：

- 简化并减少重复编码的数量。
- 减少由于重复编码所造成的错误。
- 使汇编语言程序更具可读性。

由宏可以实现的功能的例子包括：装入寄存器与执行中断的输入/输出操作，ASCII 码与二进制数据之间的转换，多字算术操作，鼠标初始化，以及串处理例行程序等。

可以按目录把宏分类列入库中，库中的宏可以用于每个程序。程序可以一次(比如初始化段地址)或多次(比如显示数据)使用宏。一个库还可以典型地包含编目的过程。

选择按宏还是按过程来编写一种操作，很大程度取决于这样一些因素：一般地说，宏执行得比较快，因为它不需要调用与返回。另一方面，过程通常产生较小的程序，因为其编码只出现一次。在一般情况下，宏是用在程序中不能多次编写且要求相当简单的地方。

这里是宏定义的基本格式：

macroname	MACRO [parameter-list]	： 定义宏	(macroname:宏名,
	[instruction]	： 宏定义体	parameter: 参数,
	ENDM	： 宏结束	instruction:指令)

在第一行的 MACRO 伪操作通知汇编程序跟在它后面的指令直到 ENDM 是宏定义的一部分。ENDM(“结束宏”)伪操作结束宏的定义。MACRO 与 ENDM 之间的指令组成宏定义体。

为了在程序中包含宏，首先要定义它或从宏库中复制它。宏定义出现在所有段的编码之

前。

21.2 简单的宏定义

首先探讨一下简单的宏定义，它为 EXE 程序初始化段寄存器：

```
INITZ MACRO           ; 宏定义
    MOV AX, @data      ; 宏
    MOV DS, AX         ; 定义
    MOV ES, AX         ; 体
ENDM                  ; 宏结束
```

这个宏的名字是 INITZ。在宏定义中被引用的数据项——@data, AX, DX, 以及 ES——必须是在程序的其他地方已经定义的，否则它们就必须是汇编程序已经知道的。

随后，你就可以在代码段中，在那些需要初始化这些寄存器的地方使用宏指令 INITZ。当汇编程序遇到宏指令 INITZ 时，要扫描它的符号指令表，并且如果未能找到该项时，检查宏指令。由于程序包含宏定义 INITZ，所以汇编程序要用产生的指令取代宏定义体——宏展开。一个程序使用宏指令 INITZ 只有一次，尽管其他的宏被设计成可以使用任意次，并且每次汇编程序都要形成宏展开。

		TITLE	A21MACR1 Simple macro definitions
		INITZ	MACRO ; 宏定义
			MOV AX, @data ; 初始化
			MOV DS, AX ; 段
			MOV ES, AX ; 寄存器
			ENDM ; 结束宏
		FINISH	MACRO ; 定义宏
			MOV AX, 4C00H ; 结束处理
			INT 21H
			ENDM ; 结束宏
		; -----	
			.MODEL SMALL
			.STACK 64
			.DATA
0000	54 65 73 74 20 6F	MESSGE	DB 'Test of macro', 13, 10, '\$'
	66 20 6D 61 63 72		
	6F 0D 0A 24		
			.CODE
0000		BEGIN	PROC FAR
			INITZ ; 宏指令
0000	B8 ---- R 1		MOV AX, @data ; 初始化段
0003	8E D8 1		MOV DS, AX ; 寄存器
0005	8E C0 1		MOV ES, AX
0007	B4 09		MOV AH, 09H ; 请求显示
0009	8D 16 0000 R		LEA DX, MESSGE ; 信息
000D	CD 21		INT 21H
			FINISH
000F	B8 4C00 1		MOV AX, 4C00H ; 结束处理
0012	CD 21 1		INT 21H
0014		BEGIN	ENDP
			END BEGIN

图 21-1 简单的汇编宏指令

同样，下面是第二个名为 FINISH 的宏定义，它处理从程序中正常退出：

```
FINISH MACRO           ; 宏定义
```

```

MOV AX, 4C00H    : 请求
INT  21H         : 结束处理
ENDM             : 宏结束

```

图 21-1 提供一个汇编程序列表，它定义并使用了 INITZ 和 FINISH。这个特殊的汇编程序版本列出了宏展开，在每条指令的左边用编号 1 来指明它是宏指令产生的。

注意，当汇编时，宏定义不能产生任何目标码，产生目标码的是宏展开。同样地，宏展开不能使像 ASSUME 或 PAGE 那样一些在宏定义中编码的伪操作打印出来。

对于只使用一次的宏，几乎没有必要去定义它，但可以把它编入库中，供所有程序使用。下一节说明如何把宏按目录列入库中，以及如何自动地把它们包括在程序中。

21.3 在宏中使用参数

为了使宏更灵活，可以在操作数中定义参数，称为哑元。下面的名为 PROMPT 的宏定义是提供给使用 INT 21H 的功能 09H 去显示信息的：

```

PROMPT    MACRO    MESSAGE                : 哑元
MOV        AH, 09H
LEA        DX, MESSAGE
INT        21H
ENDM                      : 宏结束

```

当使用这条宏指令时，必须提供信息(message)名，它所访问的数据区是用一个 \$ 符号结束的。另外，宏可以包含所用寄存器 AH 和 DX 进栈与出栈的指令。

在宏定义中的哑元告诉汇编程序把它的名字去和宏定义体中出现的任何相同名字相匹配。例如，哑元 MESSAGE 在 LEA 指令中还是一个操作数。假设程序定义名为 MESSAGE2 的提示符如下所示：

```
MESSAGE2 DB 'Enter the date as mm/dd/yy', '$'
```

现在想使用宏指令 PROMPT 显示 MESSAGE2。为此，提供名字 MESSAGE2 作为参数：

```
PROMPT    MESSAGE2
```

在宏指令中的参数(MESSAGE2)和在原来的宏定义中的哑元(MESSAGE)相匹配：

```

宏定义：      PROMPT    MACRO    MESSAGE                (哑元)
                                   |
宏指令：      PROMPT    MESSAGE2                (参数)

```

汇编程序已经使原来的宏定义中的哑元和 LEA 语句中的操作数相匹配，用宏定义中的哑元 MESSAGE 来取代宏指令的参数 MESSAGE2。汇编程序用 MESSAGE2 取代在 LEA 中出现的 MESSAGE，并用它取代任何其他地方出现的 MESSAGE。

宏定义和它的调用表示于图 21-2。程序在起点还定义了宏 INITZ 和 FINISH，并把它用在代码段中。

哑元可以包含任何有效的名字，包括寄存器名(如 CX)。在定义宏时，可以使用任何数量的哑元，哑元之间用逗号隔开，最多可达一行 120 列(取决于汇编程序版本)。汇编程序用宏

指令中的参数取代宏定义中的哑元，从左到右，一项对一项。

TITLE	A21MACR2 (EXE)	Use of parameters
INITZ	MACRO	; 宏定义
	MOV AX,@data	; 初始化
	MOV DS,AX	; 段寄存器
	MOV ES,AX	
	ENDM	; 结束宏
PROMPT	MACRO	MESSGE ; 宏定义
	MOV AH,09H	; 请求显示
	LEA DX,MESSGE	; 提示符
	INT 21H	
	ENDM	; 结束宏
FINISH	MACRO	; 宏定义
	MOV AX,4C00H	; 结束处理
	INT 21H	
	ENDM	; 结束宏

	.MODEL SMALL	
	.STACK 64	
	.DATA	
MESSG1	DB 'Name? ', '\$'	
MESSG2	DB 'Address? ', '\$'	
	.CODE	
BEGIN	PROC FAR	
	INITZ	
	PROMPT MESSG1	
	PROMPT MESSG2	
	FINISH	
BEGIN	ENDP	
	END BEGIN	

图 21-2 使用宏参数

21.4 在宏中使用注释

可以在宏定义中编写注释说明它的用途。分号或 COMMENT 伪操作指明注释行。以下是使用分号注释的例子：

```

PFOMPT MACRO MESSGE
; 这个宏允许显示信息
    MOV AH,09H      ; 请求显示
    LEA DX,MESSGE   ; 提示符
    INT 21H
ENDM

```

由于默认值只列出了产生目标码的指令，所以汇编程序在展开宏定义时，不能自动地显示注释。为了使注释在展开内出现，使用列表伪操作.LALL(“列出全部”，包括前面的点)，放在所要求的宏指令之前：

```

.LALL
PROMPT MESSAGE1

```

一个宏定义可以包含许多注释，但可能只要列出某一些而禁止列出另一些。仍然使用.LALL

把它们列出来,但在注释前加上双分号(;;),这样一来,它们就总是不能列出了。(汇编程序默认的是.XALL,它使列表中只出现生成目标码的指令)。

另一方面,可能不要列出宏展开的任何源码,特别是在程序中宏指令使用多次时更是这样。在这种情况下,编写列表伪操作.SALL(“全部禁止”),它可以减少所打印程序的大小,尽管它对所产生的目标程序的大小是没有影响的。

列表伪操作对程序自始至终保持有效,直到遇到另一个列表伪操作为止。可以把它们放在程序中去使某些宏只列出所生成的目标码(.XALL),某些宏把目标码和注释都列出(.LALL),以及某些宏使目标码和注释都禁止列出(.SALL)。MASM 6.0 引入了条目.LISTMACROALL, .LISTMACRO 和 .NOLISTMACRO,它们分别对应于.LALL, .XALL 和.SALL。

图 21-3 中的程序说明了上述特性。该程序包含早先讨论过的宏 INITZ, FINISH 和 PROMPT。代码段包含列表伪操作.SALL,禁止列出 INITZ, FINISH 和第一个 PROMPT 的展开。对于第二个 PROMPT 的使用,列表伪操作.LALL 使汇编程序列出注释和宏展开。但请注意,在 PROMPT 的宏定义中,包含双分号(;;)的注释在宏展开时并没有被列出来。

TITLE	A21MACR3 (EXE)	Use of .LALL & .SALL
INITZ	MACRO	; 宏定义
	MOV AX,@data	; 初始化
	MOV DS,AX	; 段寄存器
	MOV ES,AX	
	ENDM	; 结束宏
PROMPT	MACRO MESSGE	
;	;; 这一宏显示任何信息	
;;	;; 数据要求有\$分界符	
	MOV AH,09H	; 请求显示
	LEA DX,MESSGE	; 提示符
	INT 21H	
	ENDM	
FINISH	MACRO	; 宏定义
	MOV AX,4C00H	; 结束处理
	INT 21H	
	ENDM	; 结束宏

	.MODEL SMALL	
	.STACK 64	
	.DATA	
MESSG1	DB 'Name? ', '\$'	
MESSG2	DB 'Address? ', '\$'	

	.CODE	
BEGIN	PROC FAR	
	.SALL	
	INITZ	
	PROMPT MESSG1	
	.LALL	
	PROMPT MESSG2	
	.SALL	
	FINISH	
BEGIN	ENDP	
	END BEGIN	

图 21-3 列出与禁止宏展开

21.5 嵌套的宏

宏定义可以包含对一个或多个其他定义的(嵌套的)宏的引用。考虑以下 2 个宏定义 SET_CURSOR 和 DISPLAY:

SET_CURSOR	MACRO	ROW, COL	DISPLAY	MACRO	MESSAGE
	MOV	AX, 02H		MOV	AX, 09H
	MOV	BX, 0		LEA	DX, MESSAGE
	MOV	DH, ROW		INT	21H
	MOV	DL, COL		EDNM	
	INT	10H			
	SENDM				

名为 CURS_DISPLAY 的第三个宏可以请求 SET_CURSOR 和 DISPLAY 宏。CURS_DISPLAY 用以下方法为 2 个嵌套的宏来定义参数:

```
CURS_DISPLAY  MACRO  M_ROW, M_COL, MESSAGE
                PUSHA
                SET_CURSOR M_ROW, M_COL
                DISPLAY MESSAGE
                PCPA
                ENDM
```

可以使用常数、变量或寄存器作为行与列来编写 CURS_DISPLAY, 如:

CURS_DISPLAY	6, 15, ERROR_MSSGE	: 常数
CURS_DISPLAY	ROW, COLUMN, ERROR_MSSGE	: 变量
CURS_DISPLAY	CH, CL, ERROR_MSSGE	: 寄存器

21.6 宏伪操作

汇编程序支持许多有用的伪操作, 包括 LOCAL, PUREG, 重复, 以及条件伪操作。

21.6.1 LOCAL(局部)伪操作

某些宏要求数据项和指令标号是在宏定义自身范围内定义的。但是, 如果在同一程序中不止一次地使用这个宏, 而汇编程序对每次出现的数据项或标号都要定义的话, 那么重名会使汇编程序产生出错信息。为了确保每次所生成的名字是唯一的, 直接在 MACRO 语句之后编写 LOCAL 伪操作, 甚至要在注释之前这样做。LOCAL 的格式是:

LOCAL	localname-1, localname-2, ...
-------	-------------------------------

图 21-4 的部分程序说明 LOCAL 的使用。DIVIDE 宏的目的是使用相继减法实现除法。宏从被除数中减去除数并对商加 1, 直到被除数小于除数为止。这个宏需要 2 个标号: 对于

循环地址是 COMP，对于在完成后退过程是 OUT。COMP 和 OUT 都定义成 LOCAL，并且可以用任何有效的名字。

该程序两次使用 DIVIDE。在第一个宏展开中，产生的符号标号对于 COMP 是??0000，而对于 OUT 则是??0001。在第二个宏展开中，符号标号分别是??0002 和??0003。用这种方法，这一特性保证在程序中所生成的每个标号都是唯一的。

TITLE	A21MACR4 (EXE) Use of LOCAL		
DIVIDE	MACRO	DIVIDEND,DIVISOR,QUOTIENT	
	LOCAL	COMP	
	LOCAL	OUT	
;	AX = div'd, BX = divisor, CX = quotient		
	MOV	AX,DIVIDEND	; 设置被除数
	MOV	BX,DIVISOR	; 设置除数
	SUB	CX,CX	; 清除商
COMP:			
	CMP	AX,BX	; 被除数 < 除数?
	JB	OUT	; 是, 退出
	SUB	AX,BX	; 被除数 - 除数
	INC	CX	; 加到商上
	JMP	COMP	
OUT:			
	MOV	QUOTIENT,CX	; 存入商
	ENDM		; 结束宏
;	-----		
	DATA		
DIVIDND1	DW	150	; 被除数
DIVSOR1	DW	27	; 除数
QUOTNT1	DW	?	; 商
DIVIDND2	DW	265	; 被除数
DIVSOR2	DW	34	; 除数
QUOTNT2	DW	?	; 商
	.CODE		
	...		
	DIVIDE	DIVIDND1,DIVSOR1,QUOTNT1	
	DIVIDE	DIVIDND2,DIVSOR2,QUOTNT2	
	...		

图 21-4 使用 LOCAL 伪操作

21.6.2 包含在库中的宏

程序设计实际上是定义宏(如 INITZ, FINISH 和 PROMPT)，并且可以任意次地使用。标准的方法是在一个描述性名字(如 MACRO.LBY)下面把磁盘库中的宏进行编目。简单地把所有的宏定义收集到一个或多个存在磁盘上的文件中：

```
INITZ    MACRO
...
        ENDM
PROMPT   MACRO  MESSAGE
...
        ENDM
```

可以使用编辑程序或字处理程序编写这个文件，但要保证它是一个未格式化的 ASCII 文件。下面的例子假定该文件是在 MACRO.LBY 名字下存放的。现在程序可以使用任何已编目的宏，而不是在程序的开始编写宏定义，像以下这样使用一个 INCLUDE 伪操作：

```
INCLUDE path: \MACRO.LBY
```

汇编程序访问名为 **MACRO.LBY** 的文件而且把所有已编目的宏定义都包含到此程序中, 虽然你的程序可能只需要它们当中的一部分。汇编的列表将包含宏定义的一个副本, 对于某些汇编程序版本是在 **.LST** 文件的第 30 列用字母 **C** 指明的。

对于一个包括两遍扫视操作的汇编程序, 可以使用以下语句使 **INCLUDE** 只在第 1 遍扫视期间发生(而不是两遍扫视都发生):

```
IF1
    INCLUDE path: \MACRO.LBY
ENDIF
```

IF1 和 **ENDIF** 是条件伪操作。**IF1** 通知汇编程序只在汇编的第 1 遍扫视时访问命名的库。**ENDIF** 终止 **IF** 逻辑。宏定义的副本不再出现在列表中——时间与空间上都节省。但是, 版本 6.0 的 **MASM** 不需要涉及两遍扫视的伪操作。

INCLUDE 的位置并不是要求很严的, 但是必须出现在任何引用库项目的宏指令之前。

PURGE(清除)伪操作。**INCLUDE** 语句的执行会使汇编程序把指定库中的所有宏定义都包括在内。该库包含宏 **INITZ**、**FINISH**、**PROMPT** 和 **DIVIDE**, 但程序只需要 **INITZ** 和 **FINISH**, 那么, **PURGE** 伪操作能从当前的汇编中删除不需要的宏 **PROMPT** 和 **DIVIDE**:

```
IF1
    INCLUDE path: \MACRO.LBY      : 包含完整的库
ENDIF
PURGE  PROMPT, DIVIDE           : 删除不需要的宏
...
INITZ   ...                     : 使用余下的宏
```

PURGE 操作只用于一个程序的汇编, 并且对存放在库中的宏没有影响。

21.6.3 连接伪操作

表示 **and** 的 **(&)** 字符通知汇编程序连接文本或符号。在下面的宏中, **&** 符号便于生成 **MOVSb**、**MOVSW** 或 **MOVSD** 指令:

```
STRMOVE  MACRO  TAG
          REP  MOVSB&TAG
          ENDM
```

用户可以把这条宏指令编写成 **STRMOVE B**, **STRMOVE W**, 或 **STRMOVE D**。然后汇编程序把参数 **B**、**W** 或 **D** 和 **MOVSB** 指令连接起来, 分别产生 **REP MOVSB**、**REP MOVSW** 或 **REP MOVSD**(这个稍显平常的例子是用于说明用途的)。

21.6.4 重复伪操作

重复伪操作 **REPT**、**IRP** 和 **IRPC** 使汇编程序去重复一个语句块, 直到伪操作结束的 **ENDM** 语句为止。(MASM 6.0 引入的条目是 **REPEAT**、**FOR** 和 **FORC**, 分别对应于 **REPT**、**IRP** 和 **IRPC**。)这些伪操作不必包含在宏定义中, 但如果它们包含在里边, 就要编写一个 **ENDM** 去

终止每个重复伪操作，同时编写另一个 ENDM 去结束 MACRO 定义。

1. REPT: 重复伪操作。 REPT(或 REPEAT)伪操作使汇编程序按照表达式项目所规定的次数，一直重复到 ENDM 的语句块：

REPT	expression	(expression: 表达式)
------	------------	-------------------

第一个例子是产生 DEC 指令 4 次：

```
REP    4
      DEC SI
      ENDM
```

第二个例子把 N 值初始化为 0，然后重复产生 DB N 5 次：

```
N=    0
REPT  5
      N=N+1
      DB N
      ENDM
```

该操作产生 5 个 DB 语句：DB 1，DB 2，DB 3，DB 4 和 DB 5。使用 REPT 可以定义一个表格或部分表格。下一个例子定义宏，该宏使用 REPT 使扬声器发声 5 次：

```
BEEPSPKR    MACRO
              MOV     AH, 02H      ; 请求输出
              MOV     DL, 07      ; 发声字符
              REPT    5            ; 重复 5 次
                  INT  21H        ; 调用中断服务
              ENDM              ; 结束重复
              ENDM              ; 结束宏
```

2. IRP: 不定重复伪操作。 IRP(或 FOR)伪操作使汇编程序重复到 ENDM 为止的语句块，它的格式是

IRP	parameter, <arguments>	parameter: 参数,
	语句	argument: 自变量,
ENDM		

包含在尖括号中的自变量由任意个有效符号组成，包括串、数字或算术常数。汇编程序用每个自变量的语句替换参数。对于第一个例子，

```
IRP    N, <3, 9, 17, 25, 28>
      DB    N
      ENDM
```

汇编程序产生 DB 3，DB 9，DB 17，DB 25，DB 28。

对于第二个例子，

```
IRP    REG <AX, BX, CX, DX>
      PUSH REG
      ENDM
```

汇编程序为每个指定的寄存器产生一个 PUSH 语句。

3. IRPC: 不定重复字符伪操作。 IRPC(或 FORC)伪操作使汇编程序重复直到 ENDM 为

止的语句块。其格式是

IRP	parameter, string
	语句
ENDM	

(string: 串)

汇编程序为在串中的每个字符生成一个语句。在下面的例子中，

```
IRPC      N, 345678
    DW      N
ENDM
```

汇编程序产生 DW 3 到 DW 8。

21.6.5 条件伪操作

较早的条件伪操作的例子是使用 IF1 去包含只在汇编第 1 遍扫视期间的库项目。条件伪操作在宏定义内最有用，但它并不只限于这种用途。每个 IFnn 伪操作必须有一个配套的 ENDIF 去结束所测试的条件。一个可选的 ELSE 可以提供另一个动作。条件的处理非常像 C 语言。下面是 IF 一类条件伪操作的格式：

```
IFxx      (条件)
...
)条件
ELSE      (选项)
...
)块
ENDIF     (IF 结束)
```

遗漏 ENDIF 会产生出错信息“Undetermined conditional”(“不确定条件”)。如果汇编程序发现该条件是真，它就使直到 ELSE 的条件块生效，或者如果不存在 ELSE，则条件块直到 ENDIF 为止；如果条件为假，汇编程序将使 ELSE 之后的条件块生效，如果不存在 ELSE，就不产生任何条件块。

下面说明汇编程序是如何处理条件伪操作的：

- **IF 表达式**：如果该表达式判断为真(非零值)，则汇编在条件块内的语句。
- **IFE 表达式**：如果该表达式判断为假(零)，则汇编在条件块内的语句。
- **IF1(无表达式)**：如果处理第 1 遍扫视，则条件块内的语句生效。
- **IF2(无表达式)**：如果处理第 2 遍扫视，则条件块内的语句生效。
- **IFDEF 符号**：如果符号是在程序中定义或表示成 EXTRN，则处理在条件块内的语句。
- **IFNDEF 符号**：如果符号没有定义或没有表示成 EXTRN，则处理在条件块内的语句。
- **IFB (自变量)**：如果自变量是空白，则处理在条件块内的语句。该自变量包含在尖括号中的。
- **IFNB (自变量)**：如果自变量不是空白，则处理在条件块内的语句。该自变量包含在尖括号中的。
- **IFIDN<arg-1>, <arg-2>**：如果自变量相等，则处理在条件块内的语句。自变量都包含在尖括号中的。

- **IFDIF<arg-1>, <arg-2>**: 如果自变量不相等, 则处理在条件块内的语句。自变量包含在尖括号中的。

IF 和 IFE 可以使用关系操作符 EQ(相等), NE(不相等), LT(小于), LE(大于或等于), GT(大于), 以及 GE(大于或等于)。例如在语句中:

```
IF expression1 EQ expression2
```

下面是使用 IFNB(如果不是空白)的简单例子。INT 21H 的功能 4CH 使程序结束处理并传送返回码到 AL 中。这个例子是修改前面已用过的 FINISH 宏来提供一个返回码:

```
FINISH    MACRO    RETCODE
MOV       AH, 4CH          ; 请求结束处理
IFNB     <RETCODE>
MOV      AL, RETCODE       ; 返回码
ELSE
MOV      AL, 00H          ; 默认的退出码
ENDIF
INT      21H              ; 调用中断服务
ENDM
```

这里是使用 IFNB 的另一个例子。标准做法是为一个过程保存寄存器, 这些寄存器必须在入口用 PUSH 指令, 并在出口用 POP 指令来保存与恢复。寄存器数量与类型随过程的不同而不同。以下的宏 PUSHMAC 和 POPMAC 是处理一个或两个寄存器的, 但可以方便地扩展成包括任意数量的寄存器:

```
PUSHMAC   MACRO    REG1, REG2    POPMAC   MACRO    REG1, REG2
IFNB <REG1>
PUSH REG1
ENDIF
IFNB <REG2>
PUSH REG2
ENDIF
ENDM
IFNB <REG1>
POP REG1
ENDIF
IFNB <REG2>
POP REG2
ENDIF
ENDM
```

该宏可以按这种方法使用:

```
PUSHMAC AX, BX
POPMAC  BX, AX
```

21.6.6 EXITM 伪操作

宏定义可以包含条件伪操作, 用来测试重要的条件。如果条件为真, 则汇编程序使用 EXITM 伪操作从任何宏的进一步展开中退出:

```
IFxx      [条件]
...       (无效条件)
EXITM
...
ENDIF
```

如果汇编程序在宏指令的展开中遇到 EXITM, 则中止宏展开并在 ENDM 以后恢复处理。还可以使用 EXITM 去结束 REPT、IRP 和 IRPC 伪操作, 即使包含在宏定义之内的也是一样。

21.6.7 使用 IF 与 IFNDEF 条件的宏

在图 21-5 中的部分程序包含了名为 DIVIDE 的宏定义, 它产生一个用相继减法实现除法的例行程序。用户必须编写带参数的 DIVIDE 宏指令, 这些参数是按被除数、除数和商的次序排列的。该宏使用 IFNDEF 检查这些数据项是不是在程序中实际定义的。对于任何没有定义的数据项, 宏要增加一个名为 COUNTER 的字段 (COUNTER 可以有任意有效的名字并且是在宏定义之内临时使用的)。在检查 3 个参数之后, 宏还要检查 COUNTER 是不是非零:

```

TITLE      A21MACR5 (EXE)  Test of IF and IFNDEF
DIVIDE     MACRO  DIVIDEND,DIVISOR,QUOTIENT
LOCAL     COMP
LOCAL     OUT
CNTR      = 0
;
;      AX = div'nd, BX = div'r, CX = quot't
IFNDEF    DIVIDEND
;      被除数未定义
CNTR      = CNTR +1
ENDIF
IFNDEF    DIVISOR
;      除数未定义
CNTR      = CNTR +1
ENDIF
IFNDEF    QUOTIENT
;      商未定义
CNTR      = CNTR + 1
ENDIF
IF        CNTR
;      宏展开结束
EXITM
ENDIF
PUSHA
MOV       AX,DIVIDEND      ; 保存寄存器
MOV       BX,DIVISOR      ; 设置被除数
MOV       CX,CX           ; 设置除数
SUB       CX,CX           ; 清除商
COMP:
CMP       AX,BX           ; 被除数 < 除数?
JB        OUT             ; 是, 退出
SUB       AX,BX           ; 被除数 - 除数
INC       CX              ; 加到商上
JMP       COMP
OUT:
MOV       QUOTIENT,CX     ; 保存商
POPA
ENDM

.286
;-----
.DATA
DIVDEND   DW      150      ; 被除数
DIVISOR   DW      27      ; 除数
QUOTENT   DW      ?       ; 商
.CODE
...
.LALL
DIVIDE    DIVDEND,DIVISOR,QUOTENT
DIVIDE    DIDND,DIVISOR,QUOT
...

```

图 21-5 使用 IF 与 IFNDEF 伪操作

```

IF    COUNTER
:    宏展开结束
EXITM
ENDIF

```

如果 COUNTER 已经被设置成非零值,那么汇编程序就产生在上述代码中表明的注释,并从任何进一步的宏展开中退出(EXITM)。注意,最初的指令把 COUNTER 清除为 0,而且 IFNDEF 块也可以只把 COUNTER 置成 1 而不是使它增量。

如果条件确实通过了所有测试,那么汇编程序便产生宏展开。在代码段中,第二个 DIVIDE 宏指令包含无效的被除数和商并只产生注释。改进该宏的方法应当是去测试除数是否为非零,以及被除数与除数是否有相同的符号。为了这些目的,使用汇编指令而不是条件伪操作,因为条件的出现是在程序被执行的时候,而不是在程序被汇编的时候。

21.6.8 使用 IFIDN 条件的宏

图 21-6 的部分程序包含了名为 MOVIF 的宏定义,它产生 MOVSB 或 MOVSW,这取决于所提供的参数。用户必须编写带参数 B(字节)或 W(字)的宏指令指明 MOVIF 是变成 MOVSB,还是变成 MOVSW。在宏定义中,IFIDN 的两次出现是

```

IFIDN    <&TAG>, <B>          IFIDN    <&TAG>, <W>
REP MOVSB                      REP MOVSW
...

```

第一次是,如果编写 MOVIF B 作为宏指令,则 IFIDN 产生 REP MOVSB;第二次是,如果编写的宏指令是 MOVIF W,那么 IFIDN 产生 REP MOVSW。如果用户没有提供 B 或 W,汇编程序就产生一个注释并默认为 MOVSB(& 操作符的正常使用是在连接的情况下)。

在代码段中的 MOVIF 的 3 个例子是为了测试 B, W, 以及无效条件的。不要试图按实际情况去执行该程序,因为 DI 和 SI 必须包含用于 MOVSB 指令的适当的值。坦白地说,这个宏不是很有用,因为其目的在于以一种简单的方法来说明条件伪操作的使用。但是,现在你应当有能力开发一些属于自己的有意义的宏。

```

TITLE    A21MACR6 (EXE)  Tests of IFIDN
MOVIF    MACRO TAG
IFIDN    <&TAG>, <B>
REP MOVSB
EXITM
ENDIF
IFIDN    <&TAG>, <W>
REP MOVSW
ELSE
;    无 B或W 标志, 默认为 B
REP MOVSB
ENDIF
ENDM
; -----
;
.CODE
...
MOVIF    B
MOVIF    W
MOVIF
...

```

图 21-6 使用 IFIDN 伪操作

21.7 要 点

- 宏定义要求 **MACRO** 伪操作，一个或多个语句的块通称为体，是由宏定义产生的。**ENDM** 伪操作是结束定义用的。
- 宏指令是在程序中的宏的使用。宏指令产生的代码是宏展开。
- **.SALL**, **.LALL** 和 **.XALL** 伪操作控制在宏展开中产生的注释与目标码的列表。
- **LOCAL** 伪操作便于在宏定义内使用名字，并且必须在宏语句之后立即出现。
- 在宏定义中使用哑元，允许用户更灵活地编写参数。
- 宏库使已编目的宏可用于其他程序。
- 条件伪操作使程序能验证宏参数。

21.8 习 题

- 21-1. 在什么情况下会使用(a)宏，(b)调用过程？
- 21-2. 为一个名为 **MACARONI** 的简单宏定义编写第一和最后一行。
- 21-3. 说明宏定义体与宏展开之间的区别。
- 21-4. 什么是哑元？
- 21-5. 为以下语句编写伪操作：(a)只列出产生目标码的指令。(b)禁止宏产生的所有指令。
- 21-6. 编写两个执行乘法的宏定义：(a)**MPYBYTE** 是产生字节与字节相乘的代码；(b)**MPYWORD** 是产生字与字相乘的代码。在宏定义中，包括作为哑元的被乘数与乘数。用一个小程序(它也需要定义所要求的数据字段)测试宏的执行情况。
- 21-7. 把 21-6 题所定义的宏存入宏库中。修改程序在汇编第 1 遍扫视期间去 **INCLUDE** 库的项目。
- 21-8. 编写一个名为 **BIOSPRINT** 的宏，使用 **INT 17H** 去打印。该宏应当包括对打印机状态的测试并应提供具有任意长度可任意定义的打印行。
- 21-9. 修改图 21-4 中的宏，产生代码能在程序执行时如除数为零，则可以绕过除法。
- 21-10. 编写、汇编与测试一个程序，该程序使用名为 **MPYBYTE**、**MPYWORD** 和 **BIOSPRINT** 的宏。(a)定义两个 1 字节字段和两个 1 字的字段，并都包含数值数据。(b)使用 **MPYBYTE** 进行 1 字节字段相乘并使用 **MPYWORD** 进行 1 字字段相乘。(c)把乘积转换成 **ASCII** 格式并使用 **BIOSPRINT** 打印。
- 21-11. 根据给定的要求确定条件伪操作。处理在条件块中的语句仅当(a)自变量是空白，(b)表达式为零，(c)两个自变量相等，(d)表达式为非零。

目的：阐述汇编、连接，以及执行单个程序的程序设计技术。

22.1 引言

到本章为止，所有程序都是由一个独立的汇编模块组成的。但存在这样的可能性：这就是开发一个程序，它是由一个主程序和一个或多个单独汇编的子程序相连接而组成的。以下是把一个程序组织成子程序的原因：

- 为了语言之间的连接——例如，把易于编码的高级语言和处理效率高汇编语言组合起来。
- 为了便于大型项目的开发，在那些项目中，不同的组分别产生各自的模块。
- 为了在执行期间覆盖程序的各部分，因为程序太大了。

每个程序都是单独汇编的并产生它自己惟一的目标(.OBJ)模块。然后，连接程序把各目标模块连接成一个组合的可执行(.EXE)模块。典型的情况是，主程序是一个开始执行的程序，并调用一个或多个子程序。子程序本身又可以调用其他子程序。

图 22-1 表示的是主程序与 3 个子程序层次的 2 个例子。在(a)部分，主程序调用了程序 1、2 和 3。在(b)部分，主程序调用子程序 1 和 2，并且只有子程序 1 调用子程序 3。

有很多方法用来组织子程序，但这种组织必须能通得过汇编程序和连接程序。还必须密切注意位置，比如子程序 1 在哪里调用子程序 2，子程序 2 又在哪里调用子程序 3，而子程序 3 反过来又在哪里调用子程序 1 的。这种过程称为递归，这是可以工作的，但如果处理得不小心，可能造成重大的执行错误。

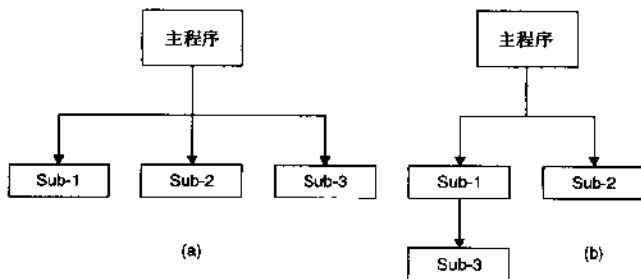


图 22-1 程序层次

22.2 段伪操作

这一节涉及用于编写 SEGMENT 伪操作的许多选项。完整的 SEGMENT 伪操作的格式是

segment-name	SEGMENT	[align] {combine} ['class']
--------------	---------	-------------------------------

下面讨论定位(align)，组合(combine)，以及类别(class)的类型。

22.2.1 定位类型

定位操作符(如果存在的话)通知汇编程序定位已命名的段，使其起始于一个特殊的存储器边界：

- **BYTE**。字节边界，对于要与另一个程序相组合的子程序的段。
- **WORD**。字边界，对于要与另一个程序相组合的子程序的段。
- **DWORD**。双字边界，通常是对于 80386 及其后继处理器的。
- **PARA**。小段边界 (能被 16 或 10H 除尽)，默认的并且是最常用的，用于主程序和子程序两者的定位。
- **PAGE**。页边界(能被 256 或 100H 除尽)。

省略第一段的定位操作符会被默认为 PARA。省略后继段的定位操作符也会默认为 PARA，如果名字是唯一的。假如名字不是唯一的，默认值就是以前所定义同名段的定位类型。

22.2.2 组合类型

组合操作符(如果存在的话)通知汇编程序和连接程序是应该把各段组合起来，还是保持它们的独立(你已经对 EXE 程序使用过 STACK 组合类型)。与本章有关的其他组合类型是 NONE，PUBLIC，以及 COMMON：

- **NONE**。该段在逻辑上是独立于其他段的，尽管它们可能都是作为物理上相邻的段结束的。这种类型是所有段伪操作的默认值。
- **PUBLIC**。连接程序把该段和其他所有被定义为 PUBLIC 并具有相同段名与类别的段组合在一起。汇编程序从第一段的起点计算偏移值。实际上，组合的段包含许多部分，每个部分都由 SEGMENT 伪操作开始并由 ENDS 结束。PUBLIC 类型是简化段伪操作的默认值。
- **COMMON**。如果 COMMON 段具有同样的名字和类别，连接程序就给它们同样的基地址。在执行期间，第二段覆盖第一段。最大的段决定公用区的长度。

22.2.3 类别的类型

已经用过类别名 ‘Stack’, ‘Data’ 和 ‘Code’。可以分配相同的类别名给有关段使汇编程序和连接程序把它们组合成一组。也就是说, 它们是一个接一个地出现的, 而不是组合在一个段里, 除非组合选项中也有 PUBLIC。类别项目可以包含任何有效的名字(包含在单引号中), 尽管对于代码段而言, 名字 ‘Code’ 才是被推荐使用的。

以下两个不相关的 SEGMENT 语句产生相同的结果, 即一个独立的定位于小段边界的代码段:

```
CODESG1 SEGMENT PARA NONE 'Code'
CODESG2 SEGMENT 'Code' (默认为 PARA 和 NONE)
```

完整的定义段的伪操作已经在第 4 章做了说明, 但以后各章用的是简化段伪操作。因为完整的段伪操作在汇编与连接子程序时, 可以提供比较严格的控制, 所以本章大多数例子都使用它们。

本章与下一章的程序举例说明许多定位, 组合与类别的选项。

22.3 段内调用

迄今为止, CALL 指令还是用于段内调用, 也就是说, 被调用的过程和调用过程是在同一个代码段中。如果被调用的过程是定义为或默认为 NEAR(即在 32K 内), 则段内调用是近调用。用存储模型定义为 Tiny, Small, 以及 Compact 的程序对于内部过程的调用默认为近调用。

近 CALL 使 IP 寄存器进入堆栈, 并用目的地址的偏移值取代 IP。因此, 近 CALL 所访问的(近)过程是在同一段内的:

	CALL nearproc	; 近调用: IP 进栈,
	...	; 连接到 nearproc
nearproc	PROC NEAR	
	...	
	RET/RETN	; 近返回: IP 出栈,
nearproc	ENDP	; 返回到调用程序

现在考虑一个近的段内 CALL 语句, 由目标码 E8 2000 组成, 其中 E8 是 CALL 的操作码, 而 2000(0020)是被调用过程的偏移地址。该操作使 IP 进入堆栈, 并把 2000 存入 IP 中成为 0020。然后处理器把在 CS 中的当前段地址和在 IP 中的偏移值组合起来(CS:IP), 以便执行下一条指令。在从被调用过程退出时, (近)RET 使存放在堆栈中的 IP 出栈并进入 IP 寄存器中, 这样, 组合了的段: 偏移地址就使控制返回到跟在 CALL 后的指令。

正如我们所讨论的, 段内转移可以是近的, 或者如果要调用的过程是在同一段内, 但是按远来定义的, 那么在这种情况下, 段内转移也可以是远的。如果 RET 是在 NEAR 过程中出现, 它就是近的。而如果 RET 是在 FAR 过程中出现的, 那它就是远的。可以把这些指令

分别编写成 RETN 或 RETF。

22.4 段间调用

如果被调用的过程是按照 FAR 或 EXTRN 定义的, 则该 CALL 被归入远调用一类, 但是, 通常在另一个代码段中却未必是这样。那些用 Medium 和 Large 存储模型定义的程序就默认为远调用。

远 CALL 首先使 CS 的内容进栈, 并在 CS 中放入一个新的段地址。然后使 IP 进栈并在 IP 中放入一个新的偏移地址 (进栈的 CS:IP 值提供紧跟在 CALL 后的指令的地址)。用这种方法, 代码段的地址和偏移地址都是为从被调用过程返回而保留的。对另一个段的调用总是段间的远调用:

	CALL farproc	: 远调用: CS 和 IP 进栈,
	...	: 连接到 farproc
farproc	PROC FAR	
	...	
	RET/RETF	: 远返回: IP 和 CS 出栈,
farproc	ENDP	: 返回到调用程序

考虑一个段间 CALL 语句, 它是由目标码 9A 0002 AF04 组成的。十六进制的 9A 是远 CALL 的操作码, 0002(或 0200)是偏移值, 而 AF04(或 04AF)是新的段地址。该操作使当前的 IP 进栈并把新的偏移值 0002 按 0200 存入 IP。接下来使当前 CS 进栈并把新的段地址 AF04 按 04AF 存入 CS。然后, 处理器把在 CS 中的当前段地址和 IP 中的偏移地址组合起来(CS:IP), 成为在被调用的子程序中要执行的第一条指令的有效地址:

在代码段中的地址:	04AF0H
在 IP 中的偏移值:	+ 0200H
有效地址:	04CF0H

在从被调用的过程退出时, 段间(远)RET 与 CALL 操作相反, 原来的 IP 与 CS 地址都出栈, 返回到各自的寄存器中。现在 CS:IP 对指向跟在原先的 CALL 之后的指令地址, 在那里继续执行程序。

近 CALL 与远 CALL 的主要区别是近 CALL 只替换 IP 偏移值, 而远 CALL 则是 CS 段地址和 IP 偏移值都要替换。近 RET/RETN 和近 CALL 相关联, 而远 RET/RETF 则是和远 CALL 相关联。

22.5 EXTRN 与 PUBLIC 属性

在图 22-2 中, 主程序(MAINPROG)调用子程序(SUBPROG)。两个模块是分别汇编的, 这是对于段间调用的要求。

在 MAINPROG 中的 CALL 必须知道 SUBPROG 存在于 MAINPROG 之外(不然的话, 汇

编程序产生一个出错信息: SUBPROG 是个未定义的符号)。伪操作 EXTRN SUBPROG: FAR 通知汇编程序任何对于 SUBPROG 的引用都附带一个 FAR 符号, 在这种情况下, 它是在另一个汇编过程中由外部定义的。由于汇编程序没有办法知道在执行时将是什么地址, 所以它在远 CALL 中产生“空”的目标码操作数(汇编的 LST 清单用 0 表示偏移值, 用短划表示段), 其后连接程序会进行填入:

```
9A 0000 ---- E          : E-外部的
```

而在 SUBPROG 中还要包含 PUBLIC 伪操作, 用来通知汇编程序和连接程序另一模块必须知道 SUBPROG 的地址。在稍后的步骤中, 当 MAINPROG 和 SUBPROG 都成功地汇编成各自的目标模块时, 就可以按下面这样连接:

```
LINK m: MAINPROG+n: SUBPROG, n: , CON
```

连接程序把在一个目标模块中的 EXTRN 和在其他目标模块中的 PUBLIC 相匹配, 并插入任何所要求的偏移地址, 然后把两个目标模块组合成一个可执行模块。如果不能匹配到引用, 则连接程序会提供出错信息, 在打算执行连接的模块之前, 应当对它进行监视。

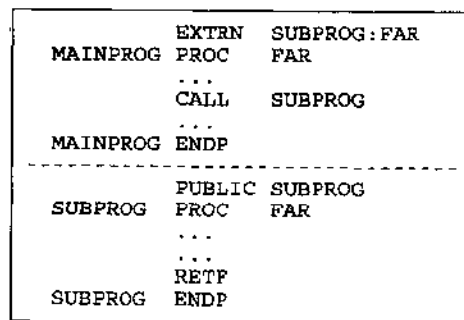


图 22-2 段间调用

22.5.1 EXTRN/EXTERN 伪操作

EXTRN 伪操作通知汇编程序: 命名的项(数据项、过程或标号)是在另一个汇编过程中定义的(MASM 6.0 引入了条目 EXTERN)。它的格式是:

EXTRN	Name: type [, ...]	(name: 名字, type: 类型)
-------	--------------------	----------------------

可以定义一个以上的名字, 直到行结束为止或用附加的 EXTRN 语句编码。而其他汇编模块又必须定义名字并把它标记为 PUBLIC。根据名字的实际定义而确定的类型项目必须是有效的。

- ABS 指出一个常数值。
- BYTE、WORD 以及 DWORD 指出数据项是一个模块所引用的, 而由另一模块所定义的。
- NEAR 和 FAR 指出过程或指令标号是一个模块所引用的, 而由另一模块所定义的。
- 用 EQU 定义一个名字。

22.5.2 PUBLIC 伪操作

PUBLIC 伪操作通知汇编程序和连接程序：在当前汇编中所定义的指定符号的地址，对其他模块是可用的。**PUBLIC** 的格式是：

PUBLIC	symbol[, ...]	(symbol: 符号)
---------------	---------------	--------------

可以定义一个以上的符号直到行结束或用附加的 **PUBLIC** 语句编码。该符号项目可以是标号(包括 **PROC** 标号)，变量，或数字。无效的项目包括寄存器名和定义值大于 2 个字节的 **EQU** 符号。

远过程的调用和 **EXTRN** 与 **PUBLIC** 的使用会有一点困难，尽管在一个模块中定义的数据要使另一个模块知道需要小心一些。

下一节探讨在程序之间使数据被知道的 2 种不同方法：使用 **EXTRN** 和 **PUBLIC**，以及传送参数。

22.6 用 EXTRN 与 PUBLIC 作为入口点

在图 22-3 中的程序由主程序 **A22MAIN1** 和子程序 **A22SUB1** 组成，它们都使用完整的段定义伪操作。主程序为堆栈、数据和代码定义段。数据段定义了 **QTY** 和 **PRICE**。代码段把 **PRICE** 装入 **AX**，把 **QTY** 装入 **BX**，然后调用子程序。在主程序中的 **EXTRN** 指出的子程序入口点是 **A22SUB1: FAR**。

子程序 **A22SUB1** 包含 **PUBLIC** 语句(在 **ASSUME** 之后)，该语句使连接程序知道它的名字是作为执行入口点的。这个子程序简单地进行 **AX** 的内容(价格)乘以 **BX** 的内容(数量)，并求出在 **DX: AX** 对中的乘积 **002E 4000H**。

由于子程序没有定义任何数据，所以不需要数据段，子程序是可以设置数据段的，但是只有它自己才能识别这些数据。

子程序使用在 **SS** 和 **SP** 中的地址，它们是由主程序发送过来的。因此，子程序不能定义堆栈，因为它所引用的堆栈是在主程序中定义的。由于连接程序要求一个 **.EXE** 程序至少要定义一个堆栈，所以主程序的堆栈就是服务于这一目的的。

现在观察一下每次汇编后的符号表。注意，主程序的符号表把 **A22SUB1** 表示成远的和外部的。子程序的符号表把 **A22SUB1** 表示成 **F**(远的)和全局的。项目全局的意思是指该名字对于 **A22SUB1** 以外的其他子程序来说是已知的。

这里是相连接模块的连接映像，它说明在存储器中的程序组织。注意，有一个堆栈和一个数据段，但两个代码段(每次汇编一个)是在不同的起始地址处，因为它们组合类型是 **NONE**。这些段是按顺序出现的，这个顺序就是用 **LINK** 命令输入它们时的顺序。在这个例子中，主程序的代码段(通常是第一个)是从偏移值 **00090H** 开始的，而子程序的代码段则是从偏移值 **000B0H** 开始的：

目标模块:		A22MAIN1+A22SUB1		
开始	停止	长度	名字	类别
00000H	0007FH	00080H	STACKSEG	STACK
00080H	00033H	00004H	DATASEG	DATA
00090H	000A5H	00016H	CODESEG	CODE
000B0H	000B2H	00003H	CODESEG	CODE

程序入口点在 0009:0000

```

                                TITLE      A22MAIN1 (EXE)  Call subprogram
                                EXTRN      A22SUB1:FAR
0000                                STACKSEG  SEGMENT PARA STACK 'Stack'
0000 0040[????]                      DW      64 DUP(?)
0080                                STACKSEG  ENDS
; -----
0000                                DATASEG   SEGMENT PARA 'Data'
0000 0140                      QTY        DW      0140H
0002 2500                      PRICE      DW      2500H
0004                                DATASEG   ENDS
; -----
0000                                CODESEG   SEGMENT PARA 'Code'
0000                                BEGIN      PROC    FAR
                                ASSUME      CS:CODESEG,DS:DATASEG,SS:STACKSEG
0000 B8 ---- R                  MOV        AX,DATASEG
0003 8E D8                      MOV        DS,AX
0005 A1 0002 R                  MOV        AX,PRICE      ; 设立价格
0008 8B 1E 0000 R              MOV        BX,QTY        ; 与数量
000C 9A 0000 ---- E            CALL       A22SUB1      ; 调用子程序
0011 B8 4C00                    MOV        AX,4C00H      ; 结束处理
0014 CD 21                      INT         21H
0016                                BEGIN      ENDP
0016                                CODESEG   ENDS
                                END          BEGIN

```

Segments and Groups:

Name	Length	Align	Combine	Class
CODESEG	0016	PARA	NONE	'CODE'
DATASEG	0004	PARA	NONE	'DATA'
STACKSEG	0080	PARA	STACK	'STACK'

Symbols:

Name	Type	Value	Attr
A22SUB1	L FAR	0000	External
BEGIN	F PROC	0000	CODESEG Length = 0016
PRICE	L WORD	0002	DATASEG
QTY	L WORD	0000	DATASEG

```

                                TITLE      A22SUB1 Called subprogram
; -----
0000                                CODESEG   SEGMENT PARA 'Code'
0000                                A22SUB1  PROC    FAR
                                ASSUME      CS:CODESEG
                                PUBLIC      A22SUB1
0000 F7 E3                      IMUL       BX          ;AX = 价格, BX = 数量
0002 CB                          RETF          ;DX:AX = 乘积
0003                                A22SUB1  ENDP
0003                                CODESEG   ENDS
                                END          A22SUB1

```

Segments and Groups:

Name	Length	Align	Combine	Class
CODESEG	0003	PARA	NONE	'CODE'

Symbols:

Name	Type	Value	Attr
A22SUB1	F PROC	0000	CODESEG Global Length = 0003

图 22-3 调用子程序: EXTRN 和 PUBLIC

对程序执行的跟踪,能揭示出 A22MAIN1 的 CS 包含 0F20[0], 并且指令 CALL A22SUB1 产生 9A 0000 200F(段地址值很可能不同)。段间 CALL 的机器码是 9AH。该操作使 IP 进栈并把 0000(CALL 的第一个操作数)存入 IP。然后使包含 0F20[0]的 CS 进栈并将 0F22[0](第二个操作数)存入 CS(寄存器内容在这里是按正常而不是相反的字节顺序排列的)。

CS:IP 对指向下一条要在 0F22[0]加上 0000 处执行的指令。在 0F220 处是什么呢? 是 A22SUB1 的入口点在它的第一条可执行指令处,是可以计算出来的。主程序是由包含 0F20[0]的 CS 开始的。根据映像,主代码段偏移值开始于偏移值 00090H,而子程序偏移值开始于偏移值 000B0H,相隔 20H 个字节。把主程序的 CS 值加上 20H 便提供了子程序代码段的有效地址:

```
A22MAIN1 的 CS 地址:      0F200H
A22MAIN1 的大小:          + 00020H
A22SUB1 的 CS 地址:      0F220H
```

装入程序确定这个地址(和我们有的一样)并替代其在 CALL 中的操作数。A22SUB1 把在 AX 和 BX 中的 2 个值相乘,产生的乘积在 DX:AX 中,并完成了-一个到 A22MAIN1 的远返回(RETf)(因为是对一个远过程的返回)。

22.7 代码段定义为 PUBLIC

图 22-4 提供的是图 22-3 的改型。在主程序中,有一个变化是 A22MAIN2,并且子程序也有一个变化是 A22SUB2,它们 2 个都涉及到代码段的 SEGMENT 伪操作中 PUBLIC 的使用:

```
CODE SEGMENT PARA PUBLIC 'Code'

                                TITLE      A22MAIN2 (EXE)  Call subprogram
                                EXTRN      A22SUB2:FAR
0000                                STACKSEG SEGMENT PARA STACK 'Stack'
0000 0040[????]                  DW        64 DUP(?)
0080                                STACKSEG ENDS

                                ; -----
0000                                DATASEG  SEGMENT PARA 'Data'
0000 0140                            QTY      DW        0140H
0002 2500                            PRICE   DW        2500H
0004                                DATASEG  ENDS

                                ; -----
0000                                CODESEG  SEGMENT PARA PUBLIC 'Code'
0000                                BEGIN
                                PROC        FAR
                                ASSUME     CS:CODESEG,DS:DATASEG,SS:STACKSEG
0000 B8 ---- R                      MOV      AX,DATASEG
0003 8E D8                          MOV      DS,AX
0005 A1 0002 R                      MOV      AX,PRICE      ; 设立价格
0008 8B 1E 0000 R                   MOV      BX,QTY        ; 与数量
000C 9A 0000 ---- E                 CALL     A22SUB2      ; 调用子程序
0011 B8 4C00                        MOV      AX,4C00H        ; 结束处理
0014 CD 21                          INT      21H
0016                                BEGIN   ENDP
0016                                CODESEG ENDS
                                END        BEGIN

-----
Segments and Groups:
  Name      Length  Align  Combine  Class
CODESEG     . . . . 0016  PARA    PUBLIC   'CODE'
```

图 22-4 调用子程序: PUBLIC 代码段

DATASEG	0004	PARA	NONE	'DATA'	
STACKSEG	0080	PARA	STACK	'STACK'	
Symbols:						
Name		Type	Value	Attr		
A22SUB2	L FAR	0000	External		
BEGIN	F PROC	0000	CODESEG	Length = 0016	
PRICE	L WORD	0002	DATASEG		
QTY	L WORD	0000	DATASEG		

		TITLE	A22SUB2 Called subprogram			
		;				
0000		CODESEG	SEGMENT PARA PUBLIC 'Code'			
0000		A22SUB2	PROC FAR			
			ASSUME CS:CODESEG			
			PUBLIC A22SUB2			
0000	F7 E3		IMUL BX		;AX = 价格, BX = 数量	
0002	CB		RETF		;DX:AX = 乘积	
0003		A22SUB2	ENDP			
0003		CODESEG	ENDS			
			END	A22SUB2		

Segments and Groups:						
Name		Length	Align	Combine	Class	
CODESEG	0003	PARA	PUBLIC	'CODE'	
Symbols:						
Name		Type	Value	Attr		
A22SUB2	F PROC	0000	CODESEG	Global Length = 0003	

图 22-4 续

有意义的结果出现在连接映像和 CALL 目标码中。在每次汇编后的符号表中, CODESEG 的组合类型是 PUBLIC, 而在图 22-3 中是 NONE。还有, 在连接映像的末端, 现在所表示的是只有一个代码段。事实上两个段有相同的名字(CODESEG), 类别('Code'), 以及 PUBLIC 属性, 使连接程序把两个逻辑的代码段组合成一个物理的代码段。下面是连接映像:

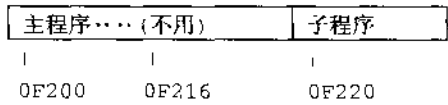
目标模块:		A22MAIN2+A22SUB2			
开始	停止	长度	名字	类别	
00000H	0007FH	00080H	STACKSEG	STACK	
00080H	00083H	00004H	DATASEG	DATA	
00090H	000B2H	00023H	CODESEG	CODE	
程序入口点在 0009: 0000					

此外, 对机器执行的跟踪表明 CALL 是远的调用, 也就是说, 虽然 CALL 是在同一段内, 但它是对 FAR 过程的调用, 其目标码是 9A 2000 200F(段地址很可能不同)。这个远 CALL 把 2000H 按 0020H 存入 IP 中, 并把 200FH 按 0F20[0]存入 CS 中。因为子程序和主程序一起共享一个公共代码段, 所以 CS 置成同样的起始地址 0F20H。但是 A22SUB2 的 CS:IP 现在提供了以下有效地址:

A22MAIN2 和 A22SUB2 的 CS 地址:	0F200H
A22SUB2 的 IP 偏移值:	+ 0020H
A22SUB2 的有效地址:	0F220H

因此, 子程序的代码段估计是从 0F220H 开始的。这正确吗? 连接映像没能清楚地指明这一点, 但可以从主程序(在偏移值 0015H 处结束)的列表中去推算这个地址(列表中表示的是 16H, 它是下一个可用单元)。由于子程序的代码段是按 PARA 定义的, 所以它开始于小段边

界(能被 10H 除尽的, 使得最右边数位是 0):



连接程序把子程序设置在紧跟主程序后的第一个小段边界处, 偏移值为 00020H。因此, 正如所计算的那样, 子程序的代码段是从 0F200H 加上 0020H, 即 0F220H 处开始的。

下一节要探讨用简化段伪操作定义的这同一个程序。

22.8 使用简化段伪操作

图 22-5 表示现在用简化段伪操作定义的以前的程序。图 22-4 是把代码段定义为 PUBLIC, 而图 22-5 默认为 PUBLIC, 这样, 两个例子都产生一个代码段。但是, 简化段伪操作的使用产生了一些重要的差别。首先, 现在段(如映像中所示)是按代码, 数据, 以及堆栈的顺序重新排列的, 尽管这不影响程序的执行。第二, 根据段表与组表, 子程序的代码段(_TEXT)定位于字(而不是小段)边界。

TITLEA22MAIN3 (EXE) Call subprogram.MODEL SMALL.STACK 64EXTRN A22SUB3:FAR;------.DATAQTY DW 0140HPRICE DW 2500H;------.CODEBEGINPROC FARMOV AX,@dataMOV DS,AXMOV AX,PRICE; 设立价格MOV BX,QTY; 与数量CALL A22SUB3; 调用子程序MOV AX,4C00H; 结束处理INT 21HENDPBEGINENDBEGIN

------.DATAQTY DW 0140HPRICE DW 2500H;------.CODEBEGINPROC FARMOV AX,@dataMOV DS,AXMOV AX,PRICE; 设立价格MOV BX,QTY; 与数量CALL A22SUB3; 调用子程序MOV AX,4C00H; 结束处理INT 21HENDPBEGINENDBEGIN

Segments and Groups:

Name	Length	Align	Combine	Class
DGROUP				GROUP
_DATA	0004	WORD	PUBLIC	'DATA'
_STACK	0040	PARA	STACK	'STACK'
_TEXT	0016	WORD	PUBLIC	'CODE'

Symbols:

Name	Type	Value	Attr
A22SUB3	L FAR	0000	External
BEGIN	F PROC	0000	_TEXT Length = 0016
PRICE	L WORD	0002	_DATA
QTY	L WORD	0000	_DATA

TITLEA22SUB3 Called subprogram.MODEL SMALL.CODEA22SUB3PROC FAR

0000A22SUB3PROC FAR

图 22-5 调用子程序: 简化段伪操作

bbs.theithome.com

0000 F7 E3		PUBLIC A22SUB3		
0002 CB		MUL BX		;AX = 价格, BX = 数量
0003	A22SUB3	RETF		;DX:AX = 乘积
		ENDP		
		END	A22SUB3	

Segments and Groups:				
Name	Length	Align	Combine	Class
DGROUP	GROUP			
_DATA	0000	WORD	PUBLIC	'DATA'
_TEXT	0003	WORD	PUBLIC	'CODE'
Symbols:				
Name	Type	Value	Attr	
A22SUB3	F PROC	0000	_TEXT Global	Length = 0003

图 22-5 续

对机器执行情况的跟踪表明 CALL 的目标码是 9A 1600 170F(段地址很可能不同)。这时, 新的偏移值是 16H, 而段地址是 0F17H。因为主程序和子程序共享一个公共代码段, 所以 CS 设置成同样的起始地址, 对于两者都是 0F17[0]。可以计算 A22SUB3 的有效地址如下:

A22MAIN3 和 A22SUB3 的 CS 地址:	F170H
A22SUB3 的 IP 偏移值:	016H
A22SUB3 的有效地址:	F186H

可以从主程序的列表中推算该地址, 主程序在偏移值 0015H 处结束 (列表中表明是 16H, 它是下一个可用单元)。因为映像表明主代码段是从 00000H 开始的, 跟着 0015H 的下一个字边界是在 00016H, A22SUB3 从那里开始, 正确地定位于字边界。下面是连接映像:

目标模块: A22MAIN3+A22SUB3				
开始	停止	长度	名字	类别
00000H	00018H	00019H	_TEXT	CODE
0001AH	0001DH	00004H	_DATA	DATA
00020H	0005FH	00040H	STACK	STACK
程序入口点在 0000: 0000				

22.9 传送参数到子程序

使被调用的子程序知道数据的一般方法是传送参数, 在这里, 程序是通过堆栈帧传送参数的。在这种情况下, 要确保每个 PUSH 引用的字(或双字)是在存储器或寄存器中。正如第 7 章所讨论过的, 程序可以用值(实际的数据项)或者用引用(该数据项的地址)来传送参数。本节给出这两种方法的例子。

堆栈帧是堆栈的一部分, 它是调用程序用来传送参数, 而被调用的子程序用来取得参数的地方。被调用的子程序还可以使用堆栈帧作为局部数据的暂存区。BP 寄存器起帧指针的作用, 而子程序则是 BP 与 SP 两个都使用。

22.9.1 用值传送参数

在图 22-6 中, 在调用子程序 A22SUB4 之前, 调用程序 A22MAIN4 使值 PRICE 和 QTY 都进入堆栈。最初, SP 包含堆栈的大小 80H。每个字进栈, SP 要减 2。

```

TITLE      A22MAIN4 (EXE)  Passing parameters
EXTRN      A22SUB4:FAR
;
STACKSEG   SEGMENT PARA STACK 'Stack'
            DW      64 DUP(?)
STACKSEG   ENDS
;
DATASEG    SEGMENT PARA 'Data'
QTY         DW      0140H
PRICE      DW      2500H
DATASEG    ENDS
;
CODESEG     SEGMENT PARA PUBLIC 'Code'
BEGIN
PROC      FAR
ASSUME CS:CODESEG,DS:DATASEG,SS:STACKSEG
MOV       AX,DATASEG
MOV       DS,AX
PUSH      PRICE           ;保留价格
PUSH      QTY             ;与数量
CALL      A22SUB4         ;调用子程序
MOV       AX,4C00H        ;结束处理
INT       21H
BEGIN     ENDP
CODESEG   ENDS
END       BEGIN
;
TITLE      A22SUB4 Called subprogram
CODESEG     SEGMENT PARA PUBLIC 'Code'
A22SUB4     PROC      FAR
ASSUME CS:CODESEG
PUBLIC A22SUB4
PUSH      BP
MOV       BP,SP
MOV       AX,[BP+8]        ;取价格
MOV       BX,[BP+6]        ;取数量
MUL       BX               ;DX:AX = 乘积
POP       BP
RET       4                ;返回调用程序
A22SUB4     ENDP
CODESEG     ENDS
END

```

图 22-6 用值传送参数

1. 第一个 PUSH 把 PRICE(2500H)存放在堆栈帧的偏移值 7EH 处。
2. 第二个 PUSH 把 QTY(0140H)存放在堆栈帧的偏移值 7CH 处。
3. CALL 使 CS 的内容(对于这次执行,它是 0F20H。)进栈到堆栈帧的 7AH 处。由于子程序是 PUBLIC 的,所以连接程序把两个代码段组合起来,而 CS 的地址对于两个代码段都是一样的。
4. CALL 还使 IP 的内容 0012H 进栈到堆栈帧的 78H 处。现在 SP 包含 78H,而堆栈帧表示如下:

偏移值	7EH	0025H	(价格) (PRICE)
	7CH	4001H	(数量) (QTY)
	7AH	200FH	(CS 的内容)
	78H	1200H	(IP 的内容)

被调用程序需要使用 BP 去取堆栈帧中的参数。它的第一个动作是为调用程序保存 BP 的内容，所以要使 BP 进栈。在这个例子中，BP 碰巧包含 0，它被 PUSH 存入堆栈帧的偏移值 76H 处。然后，程序把 SP 的内容(0076H)放入 BP，这是因为 BP(而不是 SP)是作为变址寄存器使用的。由于 BP 现在是 0076H，所以 PRICE 是在堆栈的 BP+8(偏移值 7EH)处，而 QTY 是在 BP+6(偏移值 7CH)处。因为有 3 个字(6 个字节)是在 QTY 进栈后才进栈的，所以我们知道这些相关的单元。例行程序分别把 PRICE 和 QTY 从堆栈传送到 AX 和 BX，并完成乘法运算。

在返回到调用程序之前，子程序使 BP 出栈(把 0 地址回送到 BP)，SP 加 2，从 76H 到 78H。

最后的指令 RETF 是远返回到调用程序，它完成以下操作：

- 使当时的堆栈帧顶的字(1200H)出栈到 IP，并使 SP 加 2，从 78H 到 7AH。
- 使当时的堆栈帧顶的字(0F20H)出栈到 CS，并使 SP 加 2，从 7AH 到 7CH。

由于 2 个所传送的参数是在偏移值 7CH 和 7EH 处，所以返回指令被编码为 RETF 4。立即值 4 指出所传送参数的字节数(在这种情况下是 2 个 1 字参数)。RETF 把出栈值加到 SP，把它改正为 80H。实际上，因为在堆栈中的参数已不再需要，所以操作便“丢弃”了它们并正确地返回到调用程序。注意，虽然 POP 和 RET 操作使 SP 增量，但实际上并不能抹除堆栈的内容。

这里是连接映像：

目标模块: A22MAIN4+A22SUB4				
开始	停止	长度	名字	类别
00000H	0007FH	00080H	STACKSEG	STACK
00080H	00083H	00004H	DATASEC	DATA
00090H	000BEH	0002FH	CODESEG	CODE
程序入口点在 0009:0000				

22.9.2 用引用传送参数

在图 22-7 中的程序用引用传送参数，在这种情况下，是 QTY 与 PRICE 的地址。在被调用的子程序 A22SUB5 中的主要区别是把 PRICE 的地址取到 DI 中，把 QTY 的地址取到 BX

中:

```
MOV DI, [BP+8]      ; 价格和
MOV BX, [BP+6]      ; 数量的地址
```

然后, 使用现在在 DI 中的地址得到 PRICE, 存入 AX 中:

```
MOV AX, [DI]        ; 得到价格
```

并且使用 BX 中的地址得到 QTY, 与 AX 中的 PRICE 相乘:

```
IMUL WORD PTR [BX]  ; 乘以数量
```

该程序是另一种自说明性的。

```

TITLE    A22MAIN5 (EXE)  Passing parameters
EXTRN    A22SUB5:FAR
STACKSEG SEGMENT PARA STACK 'Stack'
        DW 64 DUP(?)
STACKSEG ENDS
; -----
DATASEG  SEGMENT PARA 'Data'
QTY      DW 0140H
PRICE    DW 2500H
DATASEG  ENDS
; -----
CODESEG  SEGMENT PARA PUBLIC 'Code'
BEGIN    PROC FAR
        ASSUME CS:CODESEG,DS:DATASEG,SS:STACKSEG
        MOV     AX,DATASEG
        MOV     DS,AX
        LEA     CX,PRICE
        LEA     DX,QTY
        PUSH    CX          ; 保存价格与数量
        PUSH    DX          ; 的地址
        CALL    A22SUB5     ; 调用子程序
        MOV     AX,4C00H    ; 结束处理
        INT     21H
        BEGIN    ENDP
CODESEG  ENDS
        END      BEGIN
; -----
TITLE    A22SUB5 Called subprogram
CODESEG  SEGMENT PARA PUBLIC 'Code'
A22SUB5  PROC FAR
        ASSUME CS:CODESEG
        PUBLIC A22SUB5
        PUSH    BP
        MOV     BP,SP
        MOV     DI,[BP+8]   ; 价格与数量
        MOV     BX,[BP+6]   ; 的地址
        MOV     AX,[DI]     ; 取价格
        IMUL    WORD PTR [BX] ; 乘以数量
        POP     BP
        RETF    4           ; 返回到调用程序
A22SUB5  ENDP
CODESEG  ENDS
        END

```

图 22-7 用引用传送参数

22.10 ENTER 与 LEAVE 指令

ENTER 指令(由 80286 引入)是用于为被调用的过程(需要接受传送参数)建立临时堆栈帧

的。LEAVE 指令是和 ENTER 相对应的，用来终止堆栈帧。它们的格式是：

{label: }	ENTER	size, nest-level	(size: 大小, nest-level: 嵌套层)
{label: }	LEAVE	[no operand]	

- 大小操作数指定为堆栈帧分配操作数的字节数。虽然堆栈通常是由字或双字组成的，但 ENTER 允许传送字节大小的参数。
- 嵌套层操作数指定操作的嵌套层数。对于 BASIC, C 以及 Fortran 编译程序来说，层数总是 0。

例如，指令 ENTER 4, 0 等效于：

```
PUSH BP
MOV BP, SP
SUB SP, 4
```

该操作的效果是建立一个堆栈帧，在这个堆栈帧中 BP 指向其顶部而 SP 指向底部。程序在此堆栈帧内传送参数给被调用的过程。

LEAVE 指令和 ENTER 的操作是相反的，终止堆栈帧。LEAVE 操作等效于：

```
MOV SP, BP
POP BP
```

该操作把 SP 和 BP 恢复成原来的值。

在图 22-8 的程序中，ENTER 为把两个参数(行与列)传送给过程 SET_CURSOR 而建立了一个 2 字节的堆栈帧。注意，程序是向堆栈传送参数或从堆栈取出参数，而不是使用 PUSH 与 POP。注释表示了 SP 与 BP 的内容，假定它们分别是 40H 与 00H 开始的。

TITLE A22ENTER (EXE) Use of ENTER/LEAVE			
.MODEL SMALL			
.STACK 64			
.DATA			
ROW	EQU	5	
COL	EQU	8	
.285 ; -----			
.CODE			
BEGIN	PROC	FAR	
	MOV	AX, @data	
	MOV	DS, AX	; SP = 40, BP = 0
	ENTER	2, 0	; SP = 3C, BP = 3E
	MOV	BYTE PTR [BP-1], ROW	; 行与列
	MOV	BYTE PTR [BP-2], COL	; 进入堆栈帧
	CALL	SET_CURSOR	; SP = 3A, BP = 3E
	LEAVE		; SP = 40, BP = 0
	MOV	AX, 4C00H	; 结束处理
	INT	21H	
BEGIN	ENDP		
SET_CURSOR	PROC	NEAR	
	PUSHA		; SP = 2A, BP = 3E
	MOV	AH, 02H	; 设置光标:
	MOV	BX, 0	; 页
	MOV	DH, [BP-1]	; 行
	MOV	DL, [BP-2]	; 列
	INT	10H	; 行
	POPA		; SP = 3A, BP = 3E
	RET		; SP = 3C, BP = 3E
SET_CURSOR	ENDP		
	END	BEGIN	

图 22-8 使用 ENTER 与 LEAVE

遵循本章所讨论的通用规则，那么应该连接一个由多于 2 个汇编模块所组成的程序。但是要留心堆栈的大小：对于使用许多 PUSH 与 CALL 操作的大型程序，定义 64 个字可能是合理的预防措施。

第 23 章涉及一些有关管理存储器与执行覆盖程序的重要概念。第 25 章提供了段的一些附加特性，包括在同一汇编模块中定义一个以上的代码段或数据段，以及把它们组合成一个公共段的 GROUP 的使用。

22.11 C/C++程序与汇编语言程序的连接

虽然 C/C++ 程序可以实现许多汇编语言的功能，但还是有不少特性只有汇编语言才能做到：

- 执行 PUSH 和 POP 操作。
- 访问 BP 与 SP 寄存器。
- 初始化某些段寄存器。
- 执行时间要求严格的例程序，比如显示视频图形和实现通过端口的 I/O。

以下各节说明 C/C++ 程序与汇编语言模块接口方面的要求。该资料是基于 Microsoft C 和 C++ 编译程序的。

存储模型。调用程序与被调用的汇编程序必须是用同样的存储模型定义的。用 Tiny, Small 和 Compact 模型定义的程序产生对外部模块的近调用，它只使 IP 进栈。Medium 和 Large 存储模型定义的程序产生远调用，它使 IP 与 CS 都进栈。Flat 模型则是在 Windows 下按保护模式运行，产生近调用。

汇编 MODEL 语句指明 C/C++ 约定，比如 MODEL SMALL, C。

命名约定。汇编模块必须使用与 C/C++ 兼容的有关段与变量的命名约定。在 C/C++ 程序中的所有外部名字都包含一个前导的下划线字符，如 _column。汇编程序引用在 C/C++ 模块中的函数与变量也必须由一个下划线(_)开始。

而且，由于 C/C++ 是对大小写字母敏感的，所以汇编模块对于任何公共的变量名应当使用和 C/C++ 模块同样的字母(大写或小写)。可以强制汇编程序用以下命令行选项维护大小写字母的敏感性：

```
/mx Microsoft 5.0 和 Borland 4.0
/cx Microsoft 6.0
```

寄存器。汇编的模块必须保持在 BP, SP, CS, DS, SS, DI 和 SI 中的原先的值，即在入口使它们进栈，在出口使它们出栈。

传送参数。传送参数有 3 种方法：

1. 用值。C/C++ 调用程序传送变量的一个副本在堆栈中。被调用的汇编模块可以修改传送的值，但不能访问调用程序原来的值。如果有一个以上的参数，C/C++ 从最右边的参数开始使它们进栈。

2. 用近引用。调用程序传送数据项值的偏移地址。被调用的汇编模块假设和调用程序共

享同一个数据段。

3. 用远引用。调用程序传送段与偏移地址(段在前, 然后是偏移地址)。被调用的汇编模块假设使用与调用程序不同的数据段。被调用的程序可以使用 LDS 或 LES 指令来初始化段地址。

C/C++ 程序传送参数到堆栈中是以和其他语言相反的顺序进行的。例如, 考虑语句

```
Adds (value_1, value_2);
```

该语句先使 value2, 后使 value1 进栈, 就按那样的次序并调用 Adds。在从被调用模块返回时, C/C++ 模块(不是汇编模块)使 SP 增量去丢弃传送的参数。在被调用的汇编模块中, 用于访问 2 个传送参数的典型过程如下:

```
PUSH BP          ; 保存 BP
MOV BP, SP       ; 用 SP 地址作为基址指针
MOV DH, [BP+4]   ; 从堆栈中
MOV DL, [BP+6]   ; 取得值
...
POP BP          ; 恢复 BP
RET
```

在 PUSH BP 指令之后, 堆栈帧表现为:

value_2	BP+6
value_1	BP+4
调用程序的返回地址	BP+2
BP 偏移值	BP+0

在从被调用的模块返回时, 由于由 C/C++ 调用程序承担清除堆栈的责任, 所以发出的 RET 不带立即操作数。

数据类型的兼容性。下表表示 C/C++ 变量的类型和与其等效的汇编程序的变量类型:

C 数据类型	MASM 5.x 类型	MASM 6.0 类型
char	DB	BYTE
unsigned short/int	DW	WORD
int, short	DW	SWORD
unsigned long	DD	DWORD
long	DD	SDWORD

回送值。被调用的汇编模块对于任何回送值使用以下寄存器:

C 数据类型	寄存器
char	AL
short, near int(16bit)	AX
short, near int(32bit)	EAX
long, far(16bit)	DX: AX
long, far(32bit)	EDX: EAX

22.11.1 程序举例：C 和汇编语言连接

在图 22-9 中的例子是把一个 C 程序与一个汇编子程序连接起来,该汇编子程序的惟一目的是设置光标。C/C++ 程序通过编译产生 OBJ 模块,而汇编程序则通过汇编产生 OBJ 模块。然后,连接程序把这 2 个 OBJ 模块组合成一个 EXE 可执行模块。

C 程序定义名为 temp_row 和 temp_col 的两个项,并接受由键盘输入这些变量的行与列。该程序定义汇编子程序的名字是 set_curs。它发送作为参数的 temp_row 和 temp_col 的地址给子程序用于设置光标。调用子程序并传送参数的 C 语句是:

```
set_curs (temp_row, temp_col);
```

```
#include <stdio.h>
int main (void)
{
    int temp_row, temp_col;

    printf ("Enter cursor row: ");
    scanf ("%d", &temp_row);

    printf ("Enter cursor column: ");
    scanf ("%d", &temp_col);

    set_curs (temp_row, temp_col);
    printf ("New cursor location\n");
}

;
; 对于 C 使用 small 存储模型: 近代码, 近数据
; 使用 'standard' 段名, 以及 group 为操作

_DATA      segment word 'DATA'
row         equ     [bp+4]           ; 参数
col         equ     [bp+6]           ; (自变量)
_DATA      ends

_TEXT      SEGMENT BYTE PUBLIC 'CODE'
DGROUP     GROUP     _DATA
ASSUME     CS: _TEXT, DS: DGROUP, SS: DGROUP
PUBLIC     _set_curs
_set_curs  PROC       NEAR
    PUSH    BP           ; 调用程序的 BP 寄存器
    MOV     BP, SP       ; 指向参数

    MOV     AH, 02H       ; 请求设置光标
    MOV     BX, 0         ; 显示屏幕 0
    MOV     DH, ROW       ; 来自 BP+4 的行
    MOV     DL, COL       ; 来自 BP+6 的列
    INT     10H           ; 调用中断

    POP     BP           ; 恢复 BP
    RETF                ; 返回调用程序
_set_curs  ENDP
_TEXT     ENDS
END
```

图 22-9 连接 C 到汇编程序

进栈的值是调用程序的 BP, 返回的偏移地址, 以及 2 个被传送参数的地址。

第一个被传送的参数 `temp_row` 是在堆栈帧的偏移值 `04H` 处按 `BP+04H` 被访问。第二个被传送的参数 `temp_col` 是在偏移值 `06H` 处按 `BP+06H` 被访问。子程序使用在 `DX` 中的行与列由 `INT 10H` 设置光标。在退出时，子程序使 `BP` 出栈。`RET` 指令把控制传送回调用程序，由调用程序去清除堆栈帧。

这个普通的程序产生一个大于 `20K` 字节的模块。编译程序典型地会产生相当大的开销而不管源程序的大小如何。

其他一些 `C/C++` 版本不需要遵循这里使用的一些约定。详细资料见编译程序手册，通常是在由“接口…”或“混合语言…”标题开始的部分。

22.12 要 点

- 定位操作符通知汇编程序定位命名的段从特定存储边界开始。
- 组合操作符通知汇编程序与连接程序是否去组合段或保持它们独立。
- 分配相同的类别名给相关段会使汇编程序与连接程序把这些段组合在一起。
- 如果被调用过程是被定义为或默认为 `NEAR`(在 `32K` 内)，则段内调用是近调用。如果调用是对于一个同一段内远过程，则段内调用也可以是远调用。
- 段间调用调用在另一段中的过程，并定义为 `FAR` 或 `EXTRN`。
- 在调用子程序的主程序中，入口点被定义为 `EXTRN`；在子程序中，入口点是 `PUBLIC`。
- 为了把两个代码段连接成一个段，二者必须定义成同样的名字，同样的类别，以及 `PUBLIC` 组合类型。

22.13 习 题

22-1. 给出把程序组织成子程序的 4 个理由。

下面 3 个问题适用的 `SEGMENT` 伪操作的格式是：

```
segment-name SEGMENT [align] [combine] ['class']
```

22-2. (a)`SEGMENT` 伪操作的定位选项的默认值是什么？(b)`BYTE` 选项的作用是什么？(即汇编程序要做什么动作？)

22-3. (a)`SEGMENT` 伪操作的组合选项的默认值是什么？(b)为什么要使用 `PUBLIC` 选项？(c)为什么要使用 `COMMON` 选项？

22-4. (a)`SEGMENT` 伪操作代码段的类别选项通常是什么？(b)如果 2 个段有同样的类别但没有 `PUBLIC` 组合选项，结果是什么？(c)如果 2 个段有同样的类别，而且都有 `PUBLIC` 组合选项，结果是什么？

22-5. 说明段内调用与段间调用之间的区别。

22-6. 名为 `MAINPROG` 的程序调用一个名为 `SUBCALC` 的子程序。(a)`MAINPROG` 中的什么语句通知汇编程序名字 `SUBCALC` 是在它自己的汇编之外定义的？(b)在 `SUBCALC` 中的

什么语句使它的名字让 MAINPROG 知道?

22-7. 在 22-6 题的基础上编写一个测试程序。在 MAINPROG 中定义 3 个 DW 的数据项: STOCK_QTY(现有的库存量), UNITCOST, 以及 STOCKVALUE。MAINPROG 把全部 3 个数据项作为参数传送给 SUBCALC。SUBCALC 随后又用 STOCK_QTY 去除 STOCKVALUE, 并把商存入 UNITCOST。注意, SUBCALC 回送的计算好的价格完整地放在它的参数中。

22-8. 修改题 22-7 中的程序, 使它为堆栈帧使用 ENTER 和 LEAVE 指令。

22-9. 扩展题 22-8 使得 MAINPROG 接受来自键盘的库存量与值, 子程序 SUBBINRY 把 ASCII 量转换成二进制, 子程序 SUBCALC 计算价格, 子程序 SUBASCII 则把二进制价格转换成 ASCII, 而主程序 MAINPROG 显示结果。

目的：说明系统是如何装入程序与覆盖执行模块的。

23.1 引言

本章说明程序段前缀，程序的装入程序，程序的覆盖，以及常驻程序。介绍的操作是 INT 2FH 的功能 4A01H 多路中断和以下一些 INT 21H 的功能：

25H 设置中断向量	4AH 修改分配的存储块
31H 驻留程序	4BH 装入或执行程序
35H 取中断向量	51H 取当前 PSP 地址
48H 分配存储器	58H 取/置存储器分配策略
49H 释放已分配存储器	

23.2 程序段前缀

为了执行程序，装入程序把.COM 和.EXE 程序装入到程序段中，并在段的偏移值 00H 处建立 PSP，而且在偏移值 100H 处建立程序本身。PSP 包含以下有用的字段，按照相对位置列出如下：

00-01H INT 20H 指令(CD20H)便于返回系统。

02-03H 分配给程序的存储器最后一个小段的段地址，如 xxxx0。例如，640K 指示为 00A0H，意思是 A0000[0]。

0A-0DH 结束地址(INT 22H 的段地址)。

0E-11H Ctrl+Break 退出地址(INT 23H 的段地址)。

12-15H 危急出错退出地址(INT 24H 的段地址)。

18-2BH 默认文件处理表。

2C-2DH 程序环境的段地址。

32-33H 文件处理表长度。

34-37H 处理表远指针。

50-51H 调用 INT 21H 功能(INT 21H 和 RETF)。

5C-6BH 第一命令行自变量(FCB #1)。

6C-7FH 第二命令行自变量(FCB #2)。

80-FFH 默认 DTA(磁盘传输区)的缓冲区。

FCB(文件控制块)是存取磁盘文件的一种陈旧的方法。在 PSP 中所用的这一术语简单包含格式化了的文件说明: n: filename.ext, 其中 filename 是 8 个字节, 而 ext 是 3 个。

以下几节说明在 PSP 中的有关字段。

1. PSP 18-2BH: 默认文件处理表。在 20 字节的默认文件处理表中的每个字节指向系统表(它定义有关设备或驱动器)的一个入口。最初, 表包含 0101010002FF...FF, 其中第一个 01 指的是键盘, 第二个 01 指的是屏幕, 等等:

表项	处理设备
C1 控制台	0 键盘(标准输入)
C1 控制台	1 屏幕(标准输出)
01 控制台	2 屏幕(标准错误)
00 COM1	3 辅助的(串行口)
02 LPT1	4 标准打印机
FF 未分配	5 未分配

通常, 在 PSP 偏移值 32H 处的字包含表的长度(14H 或 20), 偏移值 34H 包含了段地址, 以 IP: CS 格式表示, 其中 IP 是 18H(在 PSP 中的偏移值), CS 是 PSP 的段地址。

需要多于 20 个打开文件的程序必须释放存储器(INT 21H 的功能 4AH), 并使用功能 67H(设置最大的处理数):

```

MOV AH, 67H          : 请求更多的处理
MOV BX, count        : 新的数(20 到 65535)
INT 21H              : 调用中断服务

```

需要的存储器总量是: 每个处理一个字节, 使数目恰好达到下一字节小段加 16 个字节。该操作在 PSP 之外建立一个新的处理表, 并修改 PSP 单元 32H 和 34H。无效操作设置进位标志并在 AX 中设置一个出错码。

2. PSP 2C-2DH: 环境的段地址。为执行而装入的程序有一个相应的环境, 由系统放在存储器中, 从程序段前面的小段边界开始。默认大小是 160 字节, 最大值是 32KB。环境包含这样一些系统命令, 如 COMSPEC, PATH, PROMPT 和 SET, 它们是能适用于程序的。

3. PSP 80-FFH: 默认的 DTA 缓冲区。装入程序用在所要求的程序名(如 MASM 或 COPY)之后键入的完整文本(如果有的话)来初始化这个区域。第一个字节包含在程序名键入后紧接着按的键(如果有的话)数, 跟着该数的是被键入的字符(如果有的话), 而后是来自前一个操作的、留在存储器中的任何无用单元。

以下 4 个例子将说明 DTA 的内容与用途。

例 1: 不带操作数的命令。假定要求执行名为 CALCIT.EXE 的程序, 用键入 CALCIT<Enter> 的办法就可以了。当程序的装入程序构造 PSP 时, 它建立默认的 DTA 为 00 0D...。第一个字节内容是在名字 CALCIT 之后键入的字节数, 不包括<Enter>字符。由于除<Enter>以外, 没有按别的键, 所以数是 0。第二个字节是 0DH, 也就是<Enter>。

例 2: 带文本操作数的命令。假定想执行一个名为 COLOR 的程序并传送一个参数“BY”, 该参数通知程序设置前景颜色为蓝(B)色和背景颜色为黄(Y)色。键入程序名后跟参数:

COLOR BY。然后，装入程序把 DTA 格式化为：

```
03 20 42 59 0D ...
```

这些字节的意思是：长度为 3 后跟一个空格，“BY”和 0DH(<Enter>)。除长度外，这个字段正好包含了在程序名 COLOR 之后所键入的内容。

例 3：带文件名操作数的命令。许多程序允许在程序名之后键入文件名。如果键入了，比如 DEL A: CALCIT.OBJ<Enter>，则 DTA 包含以下内容：

```
0D 20 41 3A 43 41 4C 43 49 54 2E 4F 42 4A 0D ...
A: C A L C I T . O B J
```

长度 13(0DH)后面跟着的正好是键入的内容，包括作为<Enter>的 0DH。

例 4：带 2 个文件名操作数的命令。考虑输入一个命令后面跟着 2 个操作数，比如

```
COPY A: FILEA.ASM C: FILEB.ASM
```

装入程序用以下内容设置 DTA：

```
80H DTA: 10 20 41 3A 46 49 4C 45 41 2E 41 53 4D 20 等...
A: F I L E A . A S M 等...
```

DTA 包含键入的字符数(10H)，空格(20)，A: FILEA.ASM C: FILEB.ASM，以及作为<Enter>的 0DH。

访问 PSP

根据确定的 PSP 地址，为处理指定的文件或进行专门的操作，可以访问 PSP。为了定位.COM 程序的 DTA，简单地在 BX、DI 或 SI 中设置 80H 并访问其内容：

```
MOV SI, 80H ; DTA 的地址
CMP BYTE PTR[SI], 0 ; 检查缓冲区(DS: SI)
JE EXIT ; 零，无数据
```

但是，.EXE 程序不能总是假定它的代码段紧跟在 PSP 之后。可以要求 INT 21H 的功能 51H 把当前 PSP 的段地址传送到 BX。

在图 23-1 的部分程序中，把所要求的文件属性设置成隐藏的(02H)。用户可以键入后跟文件名的程序名，比如 A23ATTRB n: filename.ext。程序首先确定 PSP 的地址，接下来要扫描 DTA 的<Enter>字符，并用一个十六进制零的字节取代它，建立一个 ASCIIZ 的串。然后 INT 21H 的功能 43H 使用 DTA 中的 ASCIIZ 串去改变在目录中的文件属性。

对于更为灵活的程序，用户可以键入目录路径和所要求的文件属性。

PUSH	ES	; 保存 ES
PUSH	CS	; 与 DS
MOV	AH, 51H	; 请求 PSP 地址，
INT	21H	; 传送到 BX
MOV	ES, BX	; PSP 地址进入 ES
MOV	DS, BX	; 与 DS
MOV	AL, 0DH	; 查找字符 <Enter>
MOV	CX, 21	; 字节数
MOV	DI, 82H	; 在 PSP 中的起始地址

图 23-1 设置文件属性


```

        REPNZ SCASB          ;扫描<Enter>
        JNZ  exit            ;未找到, 出错
        DEC  DI              ;找到,
        MOV  ES:BYTE PTR [DI], 0 ;用 00H 取代
        MOV  AH, 43H         ;请求
        MOV  AL, 01          ;把文件属性置成
        MOV  CX, 02          ;隐含的
        MOV  DX, 82H         ;在 PSP 中的 ASCII 串,
        INT  21H             ;按 DS:DX 寻址
        POP  DS
        POP  ES
        JC   exit            ;由 INT 引起的出错吗?
;
;exit:                                ;是, 显示信息
;
;
        A10MAIN ENDP
        END    A10MAIN

```

图 23-1 续

23.3 高端存储区

处理器使用许多地址线访问存储器。线号 A20 可以访问高端存储区 (HMA) 的 64K 空间: 从 FFFF: 10H 到 FFFF: FFFFH, 正好在 1 兆字节地址上。当处理器在实 (8086) 模式下运行时, 它通常关闭 A20 线, 以便超出这一限制的那些地址能绕过一圈到存储器的起点。打开 A20 线, 则允许寻址在 HMA 中的单元。还可以要求 CONFIG.SYS 从低端存储器到 HMA 重新定位系统文件, 从而为用户程序腾出空间。

INT 2FH (多路中断) 的服务功能之一是提供对 HMA 中可用空间的检查 (通过功能 4A01H):

```

        MOV  AX, 40A1H      ; 请求 HMA 中的空间
        INT  2FH            ; 调用多路中断

```

该操作将以下值回送到寄存器中:

BX=在 HMA 中可用的空闲字节数 (如果 COMMAND.COM 没有装入到高端, 则是零)。

ES: DI=在 HMA 中第一个空闲字节的地址 (如果系统模块没有装入到高端, 则是 FFFF: FFFF)。

23.4 存储器分配策略

INT 21H 的功能 58H 提供许多决定存储器在哪里装入程序的策略。

23.4.1 功能 5800H: 取存储器分配策略

这一操作允许查询存储器分配策略:

```

        MOV  AX, 5800H      ; 请求取策略

```

INT 21H ; 调用中断服务

该操作清除进位标志，并把策略回送到 AX：

- 00H=最先适合（默认）：从常规存储器最低地址开始查找第一个可用块，块的大小能够装入程序。
- 01H=最佳适合：查找在常规存储器中最小的可用块，该块能够装入程序。
- 02H=最后适合：从常规存储器最高地址开始查找第一个可用块。
- 40H=最先适合，只是高端：从存储器上部的最低地址开始查找第一个可用块。
- 41H=最佳适合，只是高端：在存储器的上部查找最小的可用块。
- 42H=最后适合，只是高端：从存储器上部的最高地址开始查找第一个可用块。
- 80H=最先适合，高端：从存储器上部的最低地址开始查找第一个可用块。如果没找到，再查找常规存储器。
- 81H=最佳适合，高端：从存储器上部查找最小可用块。如果没找到，再查找常规存储器。
- 82H=最后适合，高端：从存储器上部的最高地址开始查找第一个可用块。如果没找到，再查找常规存储器。

最佳与最后适合策略适用于多任务系统，可能会产生一些存储器碎片，因为各程序是同时运行的，当程序结束处理时，其存储区便释放给系统。

23.4.2 功能 5801H：设置存储器分配策略

这一操作允许改变存储器分配策略。为了设置策略，把 AL 置成代码 01 以及 BX 置成策略代码。出错时设置进位标志并回送 01（无效功能）到 AX。

23.4.3 功能 5802H：取上部存储器连接

这一操作指明程序是否可以从上部存储区(640K 以上)分配存储器。该操作清除进位标志，并回送以下代码之一到 AL：00H 的意思是该区未被连接并不能分配，01H 的意思是该区已被连接并可以分配。

23.4.4 功能 5803H：设置上部存储器连接

该操作可以连接或不连接上部存储区，并且如果该区是被连接的，则可以从它这里分配存储器：

```
MOV AX, 5803H      ; 请求
MOV BX, linkflag    ; 连接/不连接
INT 21H            ; 上部存储区
```

linkflag 操作数有以下意义：00H=不连接该区，01H=连接该区。成功的操作会清除进位标志，并允许程序从它这里分配存储器。出错时设置进位标志，并回送给 AX 代码 01（CONFIG.SYS 不包含 DOS=UMB）或 07（存储器连接被破坏）。

23.5 程序的装入程序

在装入.COM 和.EXE 程序时,程序的装入程序要在存储器中的程序段的 00H 单元建立程序段前缀,并把程序装入到 100H。除这一步骤之外,.COM 与.EXE 的装入与执行步骤是有区别的。主要区别是,当要把文件存入磁盘时,连接程序要在该.EXE 文件中插入一个专门的标题记录,而装入程序使用这一记录进行装入。

23.5.1 .COM 程序的装人与执行

在装入.COM 程序时,装入程序

- 在存储器中程序的前面安装 PSP。
- 用 PSP 的第一个字节地址设置 4 个段寄存器。
- 把堆栈指针(SP)设置到 64K 段的末端,偏移值为 FFFEh(或如果段不够大时,则设置到存储器的末端),并且使一个为零的字进栈。
- 把指令指针(IP)设置到 100H(PSP 的大小),并允许控制传送到 CS: IP 产生的地址处继续进行下去,那是紧跟 PSP 后的第一个单元,是程序的第一个字节,而且它应当包含可执行指令。

23.5.2 .EXE 程序的装人与执行

在连接程序把.EXE 模块存入磁盘时,该模块由两部分组成:包含控制与再定位信息的标题记录,以及实际的装入模块。

标题最小值是 512 个字节,并且如果有许多再定位项,那么还可以更长。标题包含有关可执行模块大小(它要被装入到存储器中),堆栈的地址,以及要被插入到未完成机器地址的再定位(或称浮动)偏移值的信息。在下表中,术语块指的是存储器中 512 个字节的区域:

00-01H 十六进制 4D5A(‘MZ’)标记一个.EXE 文件。

02-03H .EXE 文件最后块的字节数。

04-05H 包括标题在内的文件大小,以 512 个字节的块为增量。例如,如果大小是 1 025,那么这个字段包含 2,字段 02-03H 包含 1。

06-07H 再定位表项目数(见 1CH)。

08-09H 标题大小以 16 字节(小段)为增量,它帮助装入程序定位跟在标题后面的可执行模块的起点。最小数是 20H(32)($32 \times 16 = 512$ 字节)。

0A-0BH 当程序被装入时,必须驻留在该程序末端上的小段最小计数值。

0C-0DH 高/低装入程序开关。当连接时,要判定为执行而装入的程序是在低的存储器地址(通常),还是在高的存储器地址。值 0000H 指的是高的。否则,这个单元所包含的是必须驻留在被装入程序末端上的小段最大计数值。

0E-0FH 在可执行模块中堆栈段的偏移地址。

10-11H 把堆栈的大小定义为一个偏移值,当传送控制到可执行模块时,装入程序要把此偏移值插入 SP。

12-13H 检查和的值——在文件中,所有字的和(不考虑溢出)用作可能丢失数据的有效检查。

14-15H 偏移值(通常是,但不必须是 00H),它是在传送控制给可执行模块时,由装入程序插入 IP 寄存器的。

16-17H 在可执行模块内的代码段的偏移值,由装入程序插入 CS 的。该偏移值是相对于其他段的,因此如果代码段是第一个的话,偏移值将为零。

18-19H 再定位表的偏移值(见 1CH 的项)。

1A-1BH 覆盖数,其中零(通常是)的意思是.EXE 文件包含主程序。

1CH-末端 再定位表包含一个可变数的再定位项,如在偏移值 06-07H 处所指出的。标题的位置 06-07H 指明在可执行模块中要被再定位的项目数。起始于标题 1CH 的每个再定位项是由 2 字节的偏移值和 2 字节的段值组成的。

系统为环境与程序段构造存储块。以下是当装入与初始化.EXE 程序时,装入程序的执行步骤:

- 读出标题的格式化部分进入存储器中。
- 计算可执行模块的大小(在位置 04H 处的总文件大小减去在位置 08H 处的标题的大小),并在段的起点处把模块读入存储器中。
- 把再定位表项目读入工作区,并把每个项目的值加到起始段值上。
- 把 DS 与 ES 设置成 PSP 的段地址。
- 把 SS 设置成 PSP 的地址,加上 100H(PSP 的大小),加上 SS 偏移值(在 0EH 处)。另外,设置 SP 为在 10H 内的值,这是堆栈的大小。
- 设置 CS 为 PSP 地址加 100H(PSP 的大小),把在标题(在 16H 处)中的 CS 偏移值加到 CS 上。另外,用在 14H 处的偏移值设置 IP。CS:IP 对提供代码段的起始地址,实际上也是程序执行的起始地址。

在上述步骤之后,装入程序结束工作并丢弃.EXE 标题。CS 与 SS 是正确设置的,但该程序必须为它自己的数据段设置 DS 与 ES:

```
MOV  AX, datasegname      ; 设置 DS 与 ES
MOV  DS, AX               ;   为数据段
MOV  ES, AX               ;   地址
```

23.5.3 例:装入.EXE 程序

考虑以下连接程序为一个.EXE 程序产生的连接映像:

开始	停止	长度	名字	类别
00000H	0003AH	003BH	CODESEG	Code
00040H	0005AH	001BH	DATASEG	Data

续表				
00060H	0007FH	0020H	STACK	Stack
程序入口点在 0000: 0000				

映象提供 3 个段中的每段的相对单元(不是实际的)。(某些系统用名字的字母顺序排列这些段。)根据映象, 代码段(CODESEG)是从 00000H 开始的——它的相对单元是可执行模块的起点, 长度是 003BH 个字节。数据段 DATASEG 是从 00040H 开始的, 长度是 001BH。00040H 是跟在 CODESEG 之后定位于小段边界(边界可被 10H 除尽)的第一个地址。堆栈段 STACK 从 00060H 开始, 00060H 是跟在 DATASEG 之后的定位于小段边界的第一个地址。

在程序为执行而被装入之后, DEBUG 不能显示标题记录, 这是因为装入程序用 PSP 取代了标题记录。可以使用 DEBUG 的 L 命令从磁盘取一个扇区并用 D 命令把标题记录显示出来。例如, 从 CS:100 开始, 取自驱动器 A: (0), 相对扇区 3, 以及一个扇区(512 字节):
L 100 0 3 1。我们正检验的程序的标题包含以下相关信息, 按十六进制单元(数字数据是按相反字节顺序排列的)表示如下:

- 00H 十六进制 4D5A(“MZ”)。
- 02H 最后一个块的字节数: 5B00H(或 005BH)。
- 04H 包括标题的文件大小, 以 512 字节的块表示: 0200H($0002 \times 512 = 1\,024$ 个字节)。
- 06H 跟在标题的格式化部分之后的再定位表的项目数: 0100H(即 0001)。
- 08H 标题的大小, 以 16 字节增量: 2000H($0020H = 32$, 而 $32 \times 16 = 512$ 字节)。
- 0CH 装入低端存储器: FFFFH。
- 0EH 堆栈段的偏移单元: 6000H, 或 0060H。
- 10H 插入 SP 的偏移值: 2000H, 或 0020H。
- 14H IP 的偏移值: 0000H。
- 16H CS 的偏移值: 0000H。
- 18H 再定位表的偏移值: 1E00H, 或 001EH。

当 DEBUG 装入这一程序时, 各寄存器所包含的值如下:

SP=0200 DS=138F ES=138F
SS=13A5 CS=139F IP=0000

对于 EXE 模块, 装入程序把 DS 与 ES 设置成 PSP 的地址, 并把 CS、IP、SS 和 SP 设置成来自标题记录的值。让我们看一下装入程序是如何初始化这些寄存器的。

1. CS: IP 寄存器。依照 DS 寄存器, 当程序被装入时, PSP 的地址是 138F[0]H。因为 PSP 是 100H 字节长, 并且代码段是第一个(在偏移值 0 处), 所以代码段是在 PSP 后紧接着的 139F[0]H 处。可以看一看在标题中单元 16H 处的偏移值, 装入程序使用这些值去初始化 CS:

PSP 的开始地址 (见 DS):	138F0H
PSP 的长度:	+ 100H
代码段的偏移值:	+ 0H
代码段的地址:	139F0H

CS 提供了程序代码部分 (CODESEG) 的起始地址可以使用 DEBUG 显示命令 D CS:0000 观察在存储器中的程序的机器码。除了 .LST 标记为 R 的操作数之外, 该代码和汇编程序的 .LST

打印输出的十六进制部分是一样的。另外，把 IP 设置成 0000H，是来自标题中 14H 处的偏移值。

2. SS: SP 寄存器。装入程序使用在标题(0EH 处)中的 60H 值来设置在 SS 中的堆栈地址：

PSP 的开始地址 (见 DS) :	138F0H
PSP 的长度:	+ 100H
堆栈的偏移值 (见标题中的 0EH 单元):	+ 60H
堆栈地址:	13A50H

装入程序使用来自标题(在 10H 处)的 20H 初始化指向堆栈长度的堆栈指针。在这个例子中，堆栈被定义为 DW 16 DUP(?)，即 16 个 2 字节字段=32 或 20H。SP 指向当前栈顶。

3. DS 寄存器。装入程序用 DS 建立 PSP 的起点在 138F[0]。由于标题不包含程序的 DS 段地址，所以程序必须初始化 DS：

```
0004 B8 ---- R    MOV AX, DATASEG
0007 8E D8        MOV DS, AX
```

汇编程序保留下来未填入的 DATASEG 的机器地址已经变成标题中再定位表(从 1EH 处开始)的一个项。装入程序计算 DS 地址如下：

CS 地址:	139F0H
数据段的偏移值:	+ 40H (见连接映像)
DS 地址:	13A30H

DEBUG 显示完成的指令为 B8 A313，装入程序把 A313 装入 DS 成为 13A3。这些值在执行起点是：

寄存器	地址	映像偏移值
CS	139F[0]H	00H
DS	13A3[0]H	40H
SS	13A5[0]H	60H

作为练习，用 DEBUG 跟踪所连接的 EXE 程序，并注意在寄存器中改变的值得：

指令	变化的寄存器
MOV AX, DATASEG	IP 和 AX
MOV DS, AX	IP 和 DS
MOV ES, AX	IP 和 ES

DS 现在包含数据段的正确地址。试一试使用 D DS:00 去观察数据段，并使用 D SS:00 观察堆栈。

23.6 分配与释放存储器

INT 21H 服务允许分配、释放存储器和修改一个存储区的大小。你很可能为常驻程序和为执行装入的其他程序而使用这些服务程序，因为 DOS 是被设计成单用户环境的，所以一个程序为了执行而需要装入另一个程序时，必须释放它的一些存储空间。

23.6.1 INT 21H 的功能 48H: 分配存储器

为了给程序分配存储器, 需要功能 48H, 并且用所需要的小段数设置 BX:

```
MOV AH, 48H           ; 请求分配存储器
MOV BX, paragraphs    ; 小段数
INT 21H               ; 调用中断服务
```

该操作是从第一个存储块开始的, 并一步步地经过每块直到按照要求分配了足够大的空间, 通常是在存储器的高端。

如操作成功则清除进位标志, 并把所分配的存储块的段地址回送到 AX 中。如操作不成功则设置进位标志, 并把出错码(07=存储块被破坏或 08=存储器不足)回送到 AX 中, 以及向 BX 回送以小段为单位的最大可用块的大小。被破坏的存储块是指该操作找到的一个块, 该块的第一个字节不是 'M' 或 'Z'。

23.6.2 INT 21H 的功能 49H: 释放已分配的存储器

功能 49H 释放已分配的存储器, 通常用于释放常驻程序。在 ES 中装入要被返回块的段地址:

```
MOV AH, 49H           ; 请求释放已分配的存储器
LEA ES, seg-address    ; 块地址(小段的)
INT 21H               ; 调用中断服务
```

成功的操作会清除进位标志, 并在存储块的第二和第三个字节存放 00H, 意思是它不再使用。不成功的操作要设置进位标志, 并在 AX 中回送一个出错码(07=存储块被破坏, 以及 09H=无效的存储块地址)。

23.6.3 INT 21H 的功能 4AH: 修改已分配的存储块

功能 4AH 可以增加或减少存储块的大小。用为程序保留的小段数初始化 BX, 并用 PSP 的地址初始化 ES:

```
MOV AH, 4AH           ; 请求修改已分配的存储器
MOV BX, paragraphs    ; 小段数
LEA ES, PSP-address    ; PSP 的地址
INT 21H               ; 调用中断服务
```

程序可以用从 PSP 的地址中减去最后段的末地址的方法来计算它本身的大小。如果连接程序按字母顺序重新排列了段, 那必须保证使用的是最后一个段。

成功的操作会清除进位标志。不成功的操作要设置进位标志, 并在 AX 中回送一个出错码(07=存储块被破坏, 08=存储器不足, 09=无效的存储块地址)。在 BX 中, 回送最大可能有的大小(如果试图增加大小已经完成的话)。在 ES 中的不正确的地址可能造成 07 的错误。

23.7 装入或执行程序功能

现在来探讨如何使得正在执行的程序去装入并执行一个子程序。功能 4BH 使一个程序能为了执行而把子程序装入到存储器中。装入这些寄存器：

AL=下列之一的功能码：00H=装入并执行，01H=装入程序，03H=装入覆盖，05H=设置执行状态（本书不涉及）。

ES: BX=参数块的地址。

DS: DX=调用的子程序路径名的地址，是个大写字母的 ASCII 串。

这里是装入子程序的指令：

```
MOV AH, 4BH          ; 请求装入子程序
MOV AL, code         ; 功能码(只是装入)
LEA BX, para-block   ; 参数块地址
LEA DX, path         ; 路径名地址
INT 21H             ; 调用中断服务
```

无效操作设置进位标志，并在 AX 中回送一个出错码，如同图 17-1 所说明的那样。

23.7.1 AL=00H: 装入并执行

这一操作把 EXE 或 COM 程序装入到存储器，为该程序建立一个程序段前缀，并传送控制给段前缀使它执行。由于所有寄存器，包括 SS 都改变了，所以该操作不是针对初学者的。按照 ES: BX 寻址的参数块具有以下格式：

偏移值	用途
00H	在 PSP+2CH 处传送过来的环境块段地址。零地址的意思是被装入的程序继承了原有的环境。
02H	放在 PSP+80H 处的命令行的双字指针。
06H	传送在 PSP+5CH 处的 FCB #1 的双字指针。
0AH	传送在 PSP+6CH 处的 FCB #2 的双字指针。

双字指针的形式是偏移值：段地址。

23.7.2 AL=01H: 装入程序

该操作把 EXE 或 COM 程序装入存储器，并为此程序建立程序段前缀，但不能传送控制使它执行。按照 ES: BX 寻址的参数块有以下格式：

偏移值	用途
00H	在 PSP+2CH 处传送过来的环境块段地址。如果该地址为零，则被装入的程序继承了原有的环境。
02H	放在 PSP+80H 处的命令行的双字指针。

A10MAIN	PROC FAR	
	MOV AH, 4AH	; 减少存储空间
	MOV BX, STACK	; 堆栈的段地址
	ADD BX, 04	; 加小段表示的大小
	MOV CX, ES	; PSP 的段地址
	SUB BX, CX	; 从总和中减去
	INT 21H	
	JC A20ERR	; 没有足够的空间吗?
	MOV AX, @data	; 正确,
	MOV JS, AX	; 设置 DS 与 ES
	MOV ES, AX	
	MOV AH, 4BH	; 请求装入
	MOV AL, 00	; 与执行
	LEA BX, PARAREA	; COMMAND.COM
	LEA DX, PROGNAME	
	INT 21H	
	JC A30ERR	; 执行出错了吗?
	MOV AL, 00	; 正确, 没有出错码
	JMP A90XIT	
A20ERR:		
	MOV AL, 01	; 出错码 1
	JMP A90XIT	
A30ERR:		
	MOV AL, 02	; 出错码 2
A90XIT:		
	MOV AH, 4CH	; 请求
	INT 21H	; 结束处理
A10MAIN	ENDP	
	END A10MAIN	

图 23-2 续

INT 21H 的功能 4BH 在 AL 中具有代码 00 时, 处理 COMMAND.COM 的装入与执行。程序显示指定驱动器的目录项。

23.7.5 INT 21H 的功能 4DH: 取子程序回送值

该操作取回最后一个子程序的回送值, 这是该子程序由功能 4CH 或 31H 结束时所发送回来的。这些回送值是:

AH 包含子程序的结束方法, 其中 00H=正常结束, 01H=用 Ctrl+C 结束, 02=危急的设备出错, 以及 03H=由功能 31H(保留程序)结束。

AL 包含来自子程序的回送值。

23.8 程序覆盖

图 23-3 中的程序像图 23-2 的程序一样, 使用同样的服务(4BH), 但这次是只把程序装入到存储器而不执行它。该程序是由主程序 A23CALLV 和 2 个子程序 A23SUB1 和 A23SUB2 组成。A23CALLV 包含以下这些段:

```
STACK    SEGMENT PARA STACK 'Stack1'
DATASEG  SEGMENT PARA 'Data1'
CODESEG  SEGMENT PARA 'Code1'
```

```

TITLE      A23CALLV (EXE)  Call subprogram and overlay
EXTRN      A23SUB1:FAR
STACK      SEGMENT PARA STACK 'Stack1'
            DW      64 DUP(?)
STACK      ENDS
;-----
DATASEG    SEGMENT PARA 'Data1'
PARABLOK   LABEL WORD      ;参数块
            DW      0
            DW      0
FILENAME   DB      'C:\A23SUB2.EXE',0
ERRMSG1    DB      'Modify mem error'
ERRMSG2    DB      'Allocate error '
ERRMSG3    DB      'Seg call error '
DATASEG    ENDS
.286
CODESEG    SEGMENT PARA 'Code1'
A10MAIN    PROC FAR
            ASSUME CS:CODESEG,DS:DATASEG,SS:STACK
            MOV     AX,DATASEG
            MOV     DS,AX
            MOV     AX,0003H      ;设置方式与
            INT     10H          ;清除屏幕
            CALL    A23SUB1      ;调用子程序1

            MOV     AH,4AH       ;缩减存储器:
            MOV     BX,CS        ;CS的段地址
            MOV     CX,OFFSET B90END ;程序末端的偏移值
            SHR     CX,04        ;转换成小段并
            INC     CX           ;加1
            ADD     BX,CX        ;加到总的小段上
            MOV     CX,ES        ;减去
            SUB     BX,CX        ;PSP的段地址
            INT     21H          ;这一程序的大小
            JC      A20ERROR     ;若出错,退出

            MOV     AX,DS        ;为覆盖服务而
            MOV     ES,AX        ;初始化ES
            MOV     AH,48H       ;分配存储器
            MOV     BX,40        ;40小段
            INT     21H
            JC      A30ERROR     ;若出错,退出
            MOV     PARABLOK,AX  ;保存段地址

            MOV     AH,4BH       ;装入子程序2
            MOV     AL,03        ;用不执行
            LEA     BX,PARABLOK
            LEA     DX,FILENAME
            INT     21H
            JC      A50ERROR     ;若出错,退出

            MOV     AX,PARABLOK  ;交换PARABLOK
            MOV     PARABLOK+2,AX ;的两个字
            MOV     PARABLOK,20H ;设置CS偏移值为20H
            LEA     BX,PARABLOK
            CALL    DWORD PTR [BX] ;调用子程序2
            JMP     A90

A20ERROR:   LEA     BP,ERRMSG1
            CALL    B10DISPLY    ;显示信息
            JMP     A90
A30ERROR:   LEA     BP,ERRMSG2
            CALL    B10DISPLY    ;显示信息
            JMP     A90
A50ERROR:   LEA     BP,ERRMSG3
            CALL    B10DISPLY    ;显示信息
            JMP     A90
A90:        MOV     AX,4C00H     ;结束处理
            INT     21H
A10MAIN    ENDP

```

图 23-3 调用子程序与覆盖

```

B10DISPLY PROC    NEAR                ; 在入口设置 BP
                MOV    AX, DS          ; 为这一服务
                MOV    ES, AX          ; 初始化 ES
                MOV    AX, 1301H       ; 请求显示
                MOV    BX, 001EH       ; 页+属性
                MOV    CX, 16          ; 长度
                MOV    DX, 1510H       ; 行+列
                INT     10H
                RET
B90END:         NOP                    ; 程序末端偏移地址
B10DISPLY ENDP
CODESEG        ENDS
END            A10MAIN

```

```

TITLE          A23SUB1  Called subprogram
DATASEG        SEGMENT PARA 'Data2'
SUBMSG         DB      'Subprogram 1 reporting'
DATASEG        ENDS

CODESEG        SEGMENT PARA 'Code2'
A23SUB1        PROC    FAR
                ASSUME  CS:CODESEG, DS:DATASEG
                PUBLIC  A23SUB1
                PUSH    DS              ; 保存调用程序的 DS
                PUSH    ES              ; 与 ES
                MOV     AX, DATASEG
                MOV     DS, AX          ; 初始化 DS
                MOV     ES, AX          ; 与 ES
                MOV     AX, 1301H       ; 请求显示
                MOV     BX, 001EH       ; 页+属性
                LEA     BP, SUBMSG      ; 信息
                MOV     CX, 22          ; 长度
                MOV     DX, 0810H       ; 行+列
                INT     10H
                POP     ES              ; 恢复调用程序
                POP     DS              ; 的 ES与DS
                RET
A23SUB1        ENDP
CODESEG        ENDS
END

```

```

TITLE          A23SUB2  Called overlay subprogram
DATASEG        SEGMENT PARA 'Data'
SUBMSG         DB      'Subprogram 2 reporting'
DATASEG        ENDS

CODESEG        SEGMENT PARA 'Code'
A23SUB2        PROC    FAR
                ASSUME  CS:CODESEG, DS:DATASEG
                PUSH    DS              ; 保存调用程序的 DS
                MOV     AX, CS           ; 把第一个段地址
                MOV     DS, AX           ; 设置在 DS 中
                MOV     ES, AX
                MOV     AX, 1301H       ; 请求显示
                MOV     BX, 001EH       ; 页+属性
                LEA     BP, SUBMSG      ; 信息
                MOV     CX, 22          ; 长度
                MOV     DX, 1010H       ; 行+列
                INT     10H
                POP     DS              ; 恢复调用程序的 DS
                RET
A23SUB2        ENDP
CODESEG        ENDS
END

```

图 23-3 续

A23SUB1 是与 A23CALLV 一起被连接的，而且是被 A23CALLV 调用的。A23SUB1 的段是：

```

DATASEG        SEGMENT PARA 'Data2'
CODESEG        SEGMENT PARA 'Code2'

```

A23CALLV 的段被连接在前面——这是它们的类别名不同的原因：'Data1'，'Data2'，'Code1'，'Code2' 等等。下面是 A23CALLV+A23SUB1 的连接映像：

开始	停止	长度	名字	类别
00000H	0007FH	00080H	STACK	Stack1
00080H	000C2H	00043H	DATASEG	Data1
000D0H	0015FH	00090H	CODESEG	Code1
00160H	00175H	00016H	DATASEG	Data2
00180H	0019DH	0001EH	CODESEG	Code2

A23SUB2 也是被 A23CALLV 调用的，但分别连接。A23SUB2 的段是：

```
DATASEG    SEGMENT PARA    'Data'
CODESEG    SEGMENT PARA    'Code'
```

A23SUB2 的连接映像看来是这样(和一个有关没有堆栈的警告在一起)：

开始	停止	长度	名字	类别
00000H	00015H	00016H	DATASEG	Data
00020H	0003AH	0001BH	CODESEG	Code

这个程序与图 23-2 的不一样，它包含常规的段伪操作，汇编程序保留其原来的顺序。因此代码段有最后的和最高的地址。

当装入程序为执行而把 A23CALLV+A23SUB1 传送到存储器中时，A23CALLV 调用并以正常的方式执行 A23SUB1。近 CALL 正确地初始化 IP，但由于 A23SUB1 有它自己的数据段，所以它必须使 A23CALLV 的 DS 进栈，并建立自己的 DS 地址。A23SUB1 设置光标，显示信息，使 DS 出栈，并且返回到 A23CALLV。

为了使 A23SUB2 覆盖在 A23SUB1 上，A23CALLV 必须缩小其本身的存储空间，因为系统已经赋予其全部可用的存储器。对于 INT 21H 的功能 4AH 中计算程序大小的步骤是：

1. 把代码段的段地址设置在 BX 中。
2. 把代码段的大小(以小段计)加到 BX 上。
3. 从 BX 中减去 PSP 的段地址。

然后，INT 21H 的功能 48H 分配存储器提供把 A23SUB2 装入(覆盖)到 A23SUB1 顶上的空间，任意地把它设置成 40H 个小段。该操作把装入地址回送到 AX 中，在那里把 A23CALLV 存入 PARABLOK，这是功能 4BH 所用参数块的第一个字。

功能 4BH 带有在 AL 中的代码 03 时，把 A23SUB2 装入存储器。注意在数据段中的定义：C:\A23SUB2.EXE，0。功能 4BH 引用 CS 和 PARABLOK——第一个字包含被装入并覆盖的段地址，第二个字是个偏移值，在这种情况下是零。下面的图表有助于更清楚地理解这些步骤：

初始装入后		功能 4AH 缩减 存储器后		功能 48H 分配 存储器后	
000	PSP	000	PSP	000	PSP
100	A23CALLV	100	A23CALLV	100	A23CALLV
260	A23SUB1			260	A23SUB2

对于 A23SUB2 的远 CALL 要求定义为 IP: CS 的引用, 但 PARABLOK 是用的 CS: IP 的形式。因此 CS 的值被传送到第二个字, 并为 IP 把 20H 存入第一个字, 这是由于连接映像所表示的值是作为 A23SUB2 代码段的偏移值的。下一条指令把 PARABLOK 的地址装入 BX 并调用 A23SUB2:

```
LEA BX, PARABLOK          ; PARABLOK 的地址
CALL DWORD PTR [BX]       ; 调用 A23SUB2
```

注意, A23CALLV 不按在其代码段中的名字引用 A23SUB2, 因而不需要 EXTRN 语句来指定 A23SUB2。由于 A23SUB2 有自己的数据段, 所以它首先使 DS 进栈, 并初始化自己的地址。但是, A23SUB2 不是和 A23CALLV 一起连接的。因此, 指令 MOV AX, DASEG 只能用 DASEG 的偏移地址 0[0]H 来设置 AX, 而不是段地址。我们知道 CALL 是用第一个段地址设置 CS 的, (根据连接映像) 正好是数据段的地址, 所以把 CS 复制到 DS 给出了在 DS 中的正确地址。注意, 如果 A23SUB2 的代码段与数据段是按不同顺序排列的, 那么相应的编码必须是相反的。

A23SUB2 显示信息, 使 DS 出栈, 并返回到 A23CALLV。

23.9 常驻程序

许多程序被设计成在其他程序运行时是常驻在存储器中的, 并可以通过一些专用的键击启动它们的服务功能。在启动其他正常处理程序之前, 要先装入常驻程序。它们几乎都是.COM 程序并且还称为“结束但保留常驻”(TSR)程序。

编写常驻程序的容易部分是使它常驻。可以用 INT 21H 的功能 31H (保留程序) 的办法使它退出, 来代替正常的结束。该操作要求把程序的大小放在 DX 中:

```
MOV AH, 31H                ; 请求 TSR
MOV DX, prog-size          ; 程序的大小
INT 21H                    ; 调用中断服务程序
```

当你执行初始化例行程序时, 系统留出要常驻程序的存储块, 并且是把其后的程序装入到存储器的较高端。

常驻程序不容易编写的部分包括在它常驻以后对它的启动, 因为它不属于系统内部像 CLS, COPY, 以及 DIR 之类的那种程序。通常的方法是修改中断向量表, 使常驻程序中断所有键击, 按指定的键击或组合进行动作, 并且继续转到所有其他键击。结果是, 典型的 (但不是必要的) 常驻程序是由以下各部分组成:

1. 重新定义中断向量表中的部分单元。
2. 一个初始化过程只在第一次程序运行时执行, 并实现以下操作:
 - 用它自己的地址取代中断向量表中的地址,
 - 确定要保留常驻的那部分程序的大小,
 - 使用中断去通知系统结束执行当前程序, 并把程序的指定部分保留在存储器中。
3. 保留常驻的过程并被启动, 例如作专门的键盘输入或定时器时钟那样的操作。

实际上，初始化过程建立了使常驻程序工作的所有条件，而后允许它自己被删除。

常驻程序可以使用 2 个 INT 21H 功能来访问中断向量表，因为更先进的计算机会有定位于单元 0000H 为起点的，表这一点是没有保证的。

23.9.1 INT 21H 的功能 35H：取中断向量

为了取回在中断向量表中的特殊中断的地址，用所要求的中断号装入 AL：

```
MOV AH, 35H          ; 请求取向量
MOV AL, int#         ; 中断号
INT 21H              ; 调用中断服务
```

该操作把中断地址以段：偏移值的形式回送到 ES: BX。对于常规存储器，例如对 INT 09H 地址的请求，要把 00H 回送到 ES，并把 24H(36)回送到 BX。

23.9.2 INT 21H 的功能 25H：设置中断向量

为了设置一个新的中断地址，在 AL 中装入所要求的中断号和 DX 中装入新地址：

```
MOV AH, 25H          ; 请求设置中断向量
MOV AL, int#         ; 中断号
LEA DX, newaddr      ; 中断的新地址
INT 21H              ; 调用中断服务
```

该操作使用一个新地址取代中断现在的地址。但实际上，当指定的中断发生时，还是连接到常驻程序进行处理，而不是到正常的中断地址。

23.9.3 常驻程序举例

在图 23-4 中的名为 A23RESID 的常驻程序在按 ESC 键时会发出嘟嘟声。程序仅有的实际用途是用可能最简单的方法说明常驻程序的技术细节，使你能尝试其他类型较为有用的程序。

有关常驻程序的以下几点是有意义的：

CODESEG 开始了 A23RESID 的代码段。第一条可执行指令 JMP B10INIT 越过常驻部分把执行传送到靠近末端的 B10INIT。这个例行程序首先使用 CLI 防止任何可能碰巧在此时发生的进一步中断。然后使用 INT 21H 的功能 35H 确定在中断向量表中的 INT 09H 的地址。该操作把地址回送到 ES: BX 中，B10INIT 例行程序把它们存入 SAVEINT9。接下来，功能 25H 把程序本身的地址设置到中断向量表的 INT 09H 处，这一地址即 A10TEST，它是常驻程序的入口点。实际上，该程序保留 INT 09H 的地址，并用该程序自己的地址取代它。最后一步是确定常驻部分(直到 B10INIT 的全部代码)的大小放在 DX 中，并使用 INT 21H 的功能 31H(结束但保留常驻)退出。从 B10INIT 到末端的代码被下一个为执行而装入的程序所覆盖。

A10TEST 是这个常驻过程的名字，在用户按键时被启动。系统把执行传送到在中断向量表中的 INT 09H 的地址，它已经修改成 A10TEST 的地址。因为可能碰巧遇上中断（例如当

用户正在执行任何一类操作时)，所以 A23RESID 必须保存它使用的寄存器。程序访问键盘标志，以确定是否按了 Esc 键(扫描码 01)。如果是，程序就使扬声器发声(扬声器的使用在第 24 章的“发声”一节说明)。最后的指令包括恢复进栈的寄存器（按相反的顺序）并转移到 SAVEINT9，那里有原来的 INT 09H 的地址。现在控制被释放，返回到中断。

TITLE	A23RESID (COM)	Resident program: Beep if use
;		Esc key
CODESEG	SEGMENT PARA	
	ASSUME CS:CODESEG	
	ORG 100H	
BEGIN:	JMP B10INIT	; 转移到初始化
SAVEINT9	DD ?	; INT 09H 的地址
DURATION	DW 100H	;
A10TEST:	PUSH AX	; 保存寄存器
	PUSH CX	
	IN AL, 60H	; 从端口取得键击
	CMP AL, 01	; 扫描码是 01 (Esc) 吗?
	JNE A50EXIT	; 否, 退出
	IN AL, 61H	; 取端 L1 状态
	PUSH AX	; 并保存
	OR AL, 00000011B	; 接通扬声器
	OUT 61H, AL	
	MOV CX, 512H	; 长度
A20:	LOOP A20	;
	OR AL, 00000010B	; 将位 1 置 1
	MOV CX, 512H	;
A30:	LOOP A30	;
	POP AX	; 端口状态
	AND AL, 11111100B	; 关闭扬声器
	OUT 61H, AL	
A50EXIT:	POP CX	; 恢复寄存器
	POP AX	
	JMP CS:SAVEINT9	; 恢复 INT 09H
B10INIT:		; 初始化:
	CLI	-----
	MOV AH, 35H	; 防止中断
	MOV AL, 09H	; 取 INT 09H 的地址,
	INT 21H	; 存入 ES:BX
	MOV WORD PTR SAVEINT9, BX	; 并保存它
	MOV WORD PTR SAVEINT9+2, ES	
	MOV AH, 25H	
	MOV AL, 09H	; 为 INT 09H
	MOV DX, OFFSET A10TEST	; 设置新地址为 A10TEST
	INT 21H	
	MOV AH, 31H	; 请求保存常驻
	MOV DX, OFFSET B10INIT	; 设置大小
	STI	; 恢复中断
	INT 21H	
CODESEG	ENDS	
	END BEGIN	

图 23-4 常驻程序

下一个例子将有助于对过程的清楚了解。首先，这是一个没有 TSR 阻止中断的常规操作的说明：

1. 用户按一个键，键盘发送 INT 09H 给 BIOS。
2. BIOS 使用在中断向量表中 INT 09H 的地址确定 BIOS 例行程序。
3. 然后把控制传送给 BIOS 例行程序。
4. 例行程序取得字符，并且如果是标准字符，就把它传送到键盘缓冲器。

下面是常驻程序的过程：

1. 用户按一个键，并且键盘发送 INT 09H 给 BIOS。
2. BIOS 使用在中断向量表中 INT 09H 的地址确定 BIOS 例行程序。
3. 但现在的表包含的是在常驻程序中的 A10TEST 的地址，要把控制传送给它。
4. 如果字符是 Esc，则程序使扬声器发声。
5. A10TSET 用转移到原先保存的 INT 09H 地址的办法退出，在那里把控制传送给 BIOS 例行程序。
6. BIOS 例行程序取得字符，并且如果是一个标准字符，就把它传送到键盘缓冲器。

试一试使用 DEBUG 去检验这一程序的执行结果。使用 D 0:20 显示中断向量表在 20H(36) 处的内容，其中存放的是 INT 09H 的中断地址。第一个字是偏移值，而第二个字是段地址，它们都是按相反字节顺序存放的。例如，如果存放的地址是 0701 EF05，则使用 D 107:05EF 去观察所存放地址的内容。显示应该从 5051 1EB8 开始，那里是在常驻程序中 A10TEST 的机器码的起点。

某些程序还替换了 INT 09H 的表地址，它们不允许像这个程序那样同时使用常驻程序。

23.10 要 点

- 连接程序产生的 .EXE 模块由包含控制与再定位信息的标题记录和实际装入的模块组成。
- 在装入 .COM 程序时，装入程序用 PSP 的地址设置段寄存器，把堆栈指针设置到该段的末端，使一个零字进栈，设置指令指针为 100H(PSP 的大小)。然后，控制进行到由 CS:IP 产生的地址，即紧跟 PSP 的第一个单元。
- 在装入 .EXE 程序时，装入程序把标题记录读入到存储器中，计算可执行模块的大小，以及把模块读入到存储器中的起始段处。装入程序把每个再定位表项目的值加到起始段的值上。它把 DS 和 ES 设置成 PSP 的段地址；设置 SS 为 PSP 的地址加上 100H，再加上 SS 偏移值；设置 SP 为堆栈的大小，以及设置 CS 为 PSP 的地址加上 100H，再加上在标题中的 CS 偏移值。装入程序用在 14H 处的偏移值设置 IP。CS:IP 对提供代码段的起始地址用于程序的执行。
- PSP 的字段包括在 5CH 处的参数区 1，在 6CH 处的参数区 2，以及在 80H 处的默认磁盘传送区。
- 在启动其他正常处理的程序之前要装入常驻程序。用 INT 21H 的功能 31H 退出，这需要在 DX 中的程序的大小并留出程序常驻的存储块。

23.11 习 题

- 23-1. (a) 系统把程序段前缀存放在什么地方？(b) 它的大小是多少？
- 23-2. 在程序段前缀中这些字段的用途是什么？(a) 包含 CD20H 的第一个字节，(b) 18-2BH，

默认的文件处理表, (c)2C-2DH, 程序环境的段地址, (d)80-FFH, 默认的 DTA。

23-3. 为执行而被装入的一个.COM 程序, 它的 PSP 从单元 3AB6[0]H 开始。程序的装入程序存放在以下每个寄存器中的地址是什么? (a)CS, (b)DS, (c)ES, (d)SS。

23-4. 一个.EXE 程序的连接映像表示如下:

开始	停止	长度	名字	类别
00000H	00040H	00050H	STACK	STACK
00050H	0007EH	0002CH	CODESEG	CODE
00080H	000ACH	0002DH	DATASEG	DATA

装入程序装入该程序, 其 PSP 从单元 2AC6[0]H 开始。表现出计算结果是适当的, 请确定在装入时各寄存器的内容: (a)SS, (b)SP, (c)CS, (d)DS, (e)ES。

23-5. 常驻程序通常阻止键盘输入。这个被阻止的地址正好是什么? 在什么地方?

23-6. 以怎样有效的两种方法去为结束一个与普通程序有别的常驻程序进行编码?

第七部分

参考章节

BIOS 数据区、中断和端口

第 24 章

24

目的：描述 BIOS 数据区，中断服务和端口操作。

24.1 引言

BIOS 包含输入输出例行程序的扩展集以及系统设备状态表。操作系统和用户程序都可以请求 BIOS 例程与系统连接的设备进行通信。和 BIOS 接口的方法是调用软件中断。这一章介绍 BIOS 支持的数据区、中断程序和 00H 到 1BH 的 BIOS 中断、DOS 中断 21H 以及系统端口。

24.2 引导过程

在 PC 机上，ROM 驻留在 FFFF0H 开始的位置。开机后，处理器进入重置状态，设置所有的存储器单元为 0，对存储器执行奇偶校验，以及设置 CS 为 FFFF[0]H，IP 为 0。所以第一步要执行的指令在 FFFF:0，它指向 BIOS 的入口。BIOS 也在 40[0]:72H 存储了一个值 1234H 来通知后继操作 Ctrl+Alt+Del(重启动)不执行加电自检。

BIOS 检验各个端口来识别和初始化连接的设备，包括 INT 11H(确定设备)和 INT 12H(确定存储器大小)。然后，在常规存储器的单元 0 开始，BIOS 建立包含有中断例程地址的中断向量表。

紧接着，BIOS 确定包含系统文件的磁盘是否存在，如果有，将执行 INT 19H 存取第一个磁盘扇区，该扇区含有引导程序加载程序。这个程序是一个临时的操作系统，在它加载存储器之后，BIOS 例程把控制传递给操作系统。引导程序只有一个任务：把操作系统的第一部分装入内存。

24.3 BIOS 数据区

BIOS 在段地址 40[0]H 开始的低地址存储器保留了自己的 256 字节(100H)的数据区，该

数据区包含有按反序字节排列的数据字段，它们按照偏移地址排列。

1. 串行端口数据区

00H~07H 4 个字，4 个串行端口 COM1~COM4 的地址。

2. 并行端口数据区

08H~0FH 4 个字，4 个并行端口 LPT1~LPT4 的地址。

3. 系统设备数据区

10H~11H 设备状态，已安装设备的状态的原始指示。可以调用 INT 11H，它在 AX 中返回下列信息：

位	设备
15, 14	连接的并行端口数
11~9	连接的 RS232 串行端口数
7, 6	软盘设备数，位为 00=1，01=2，10=3，11=4
5, 4	初始化显示方式，位值为 00=未用，01=40×25 彩色，10=80×25 彩色，11=80×25 黑白。
2	指示设备(鼠标)，1=已安装
1	1=数字协处理器已存在
0	1=软盘驱动器已存在

4. 混合数据区

12H 厂商的测试标志。

5. 存储器大小数据区

13H~14H 系统板上的内存总数(千字节)。

15H~16H 扩展内存总数(千字节)。

6. 键盘数据区 1

17H 当前 SHIFT 状态的第一字节：

位	动作	字节	动作
7	Insert 激活	3	右 Alt 按下
6	CapsLock 激活	2	右 Ctrl 按下
5	NumLock 激活	1	左 shift 按下
4	ScrollLock 激活	0	右 shift 按下

“激活”表示键已经按下和设置。“按下”表示在 BIOS 存储状态时键已被按下。

18H 当前 SHIFT 状态的第二字节：

位	动作	字节	动作
7	Insert 按下	3	Ctrl/NumLock 按下
6	CapsLock 按下	2	SysReq 按下
5	NumLock 按下	1	左 Alt 按下
4	ScrollLock 按下	0	左 Ctrl 按下

19H ASCII 字符的交替键盘入口。

1AH~1BH 键盘缓冲区的头指针。

1CH~1DH 键盘缓冲区的尾指针。

1EH~3DH 键盘缓冲区(32 字节)。

7. 软盘驱动器数据区

3EH 磁盘搜索状态。位 0 对应驱动器 A, 1 为驱动器 B, 2 为驱动器 C, 3 为驱动器 D。
A 位值为 0 表示下一次搜索将重新定位于柱面 0, 以重新校准驱动器。

3FH 磁盘电机状态。若位 7=1, 则程序正在执行写操作。位 0 对应驱动器 A, 1 为 B,
2 为 C, 3 为 D。

40H 电机超时计数, 直到电机关闭

41H 磁盘状态, 指出上一次软盘驱动器操作的错误:

00H	无错误	09H	试图使 DMA 超过 64K 限制
01H	无效磁盘参数	0CH	未找到媒体类型
02H	未找到地址标志	10H	读时 CRC 出错
03H	写保护错	20H	控制器错
04H	未找到扇区	40H	搜索失败
06H	软盘改变线激活	80H	驱动器未准备好
08H	DMA 过速		

42H~48H 软盘驱动控制器状态。

8. 视频数据区 1

49H 当前显示方式, 由 1 位来指示:

位	方式	位	方式
7	单色	3	80×25 彩色
6	640×200 单色	2	80×25 单色
5	320×200 单色	1	40×25 彩色
4	320×200 彩色	0	40×25 单色

4AH~4BH 屏幕上的列数。

4CH~4DH 显示页缓冲区的大小。

4EH~4FH 显示缓冲区的起始偏移地址。

50H~5FH 8 个字, 8 页(0~7)中的每一页的当前开始位置。

60H~61H 光标开始和结束行。

62H 当前显示页。

63H~64H 当前显示的端口地址(单色为 03B4H, 彩色为 03D4H)。

65H 显示方式控制寄存器的当前设置。

66H 当前彩色调色板。

9. 系统数据区

67H~68H 数据边界时间计数。

69H~6AH 循环冗余校验(CRC)寄存器。

6BH 最后输入值。

6CH~6FH 定时器, 用反序字节计时。每秒更新 18.2 次(约 55 毫秒 1 次)。将定时器的值除以 18.2 即是从午夜开始的秒数。

70H 定时器溢出(若计时器超过午夜, 则为 1)。

71H Ctrl+Break 键设置位 7 为 1,

72H~73H 内存重置标志, 如果内容是 1234H, 按下 Ctrl+Alt+Del 则重启。

10. 硬盘数据区

74H 最后一个硬盘操作的状态(详见第 19 章)。

75H 连接的硬盘数。

11. 超时数据区

78H~7BH 并行端口超时(LPT1~LPT4)。

7CH~7FH 串行端口超时(COM1~COM4)。

12. 键盘数据区 2

80H~81H 键盘缓冲区开始的偏移地址。

82H~83H 键盘缓冲区结束的偏移地址。

13. 视频数据区 2

84H 屏幕的行数(-1)。

85H~86H 字符高度, 以扫描行计。

87H 视频信息, 按位表示:

位 0=1, 光标能以文本方式模拟

位 3=1, 视频子系统未激活

位 6~5, 视频存储器容量(11=256K 或更多)

位 7, 与传递到 INT 10H 的功能 00 的显示方式码的第 7 位相同

88H 混合视频信息。

89H 混合标志, 按位表示:

位 0=1, 表示 VGA 是激活的

位 1=1, 表示允许灰度求和

位 2=0, 表示彩色监视器, =1 表示单色监视器

位 3=1, 表示禁止加载默认调色板

位 4 和 7 为文本方式定义扫描行

位 4	位 7	模式
0	0	350 行
0	1	400 行
1	0	200 行

14. 软盘/硬盘数据区

8BH~95H 控制器和错误状态。

15. 键盘数据区 3

96H 键盘方式状态和键入标志:

位	动作	位	动作
7	顺次读 ID	3	右 Alt 按下
6	最后的代码是 ACK	2	右 Ctrl 按下
5	如果读 ID 和 KBX 强制 NumLock	1	最后的扫描码是 E0
4	安装扩展键盘	0	最后的扫描码是 E1

97H 键盘 LED 标志(位 0=ScrollLock, 位 1=NumLock, 位 2=CapsLock)。

16. 实时钟数据区

98H~A7H 等待标志的状态。

17. 保存指针数据区

A8H~ABH 指向 BIOS 存储区的指针。第一个地址(偏移量:段)指向视频保存指针表。

18. 混合数据区 2

ACH~FFH 系统保留, 为内部使用。

下列 BIOS 区域在高地址内存区:

A0000~AFFFF	视频显示图形缓冲区
B0000~B0FFF	单色视频缓冲区
B8000~B0FA0	彩色文本视频缓冲区
F0000~FFFFFF	ROM BIOS 信息区

24.4 中断服务

中断操作使一个执行程序挂起, 这样系统就能采取一些特定的动作。执行中断例程, 通

常返回控制给被中断的程序，使其继续执行。BIOS 处理 INT 00H~1FH，DOS 处理 INT 20H~3FH。

24.4.1 中断向量表

当计算机加电后，系统在常规内存 000H~3FFH 位置建立中断向量表。表中提供了 256(100H)种中断，每种中断与相关的 4 字节地址以 IP:CS 的形式表示偏移量:段。中断指令的操作数，如 INT 05H，标识了请求的类型。因为有 256 项，每一项长度为 4 字节，所以中断向量表占据了存储器开始的 1 024 个字节，从 00H 到 3FFH。表中的每个地址与一个特定中断类型的 BIOS 或者 DOS 例程相关。因此字节 0~3 含有中断 0 的地址，4~7 含有中断 1 的地址，以此类推。下面列出了一些中断操作：

中断	操作	中断	操作
0	除数为 0	14	串行端口中断
1	单步处理	16	键盘中断
2	非屏蔽中断(NMI)	17	打印机中断
3	断点地址	19	引导程序加载器
4	溢出	1A	日时钟
5	打印屏幕	1B	键盘 break 控制
8	定时器间隔	1C	定时器中断控制
9	BIOS 键盘中断	1D	视频表地址
E	磁盘中断	1E	磁盘表地址
10	视频中断	1F	ASCII 字符地址
11	设备检验	21	DOS 中断
12	内存检验	33	鼠标中断
13	磁盘 I/O		

24.4.2 执行中断

中断将 Flags 寄存器、CS 和 IP 的内容压入堆栈。例如，按下 Ctrl+PrintScrn 使 BIOS 调用 INT 05H 的向量表地址，这个地址是 0014H(05H×4=14H)。操作从 0014H 位置开始取 4 个字节地址，并在 IP 中存放 2 个字节，在 CS 中存放 2 个字节。CS:IP 中的地址指向 BIOS 区中一个例程的开始，此例程是打印视频显示区。中断通过 IRET 指令返回，从堆栈弹出 IP、CS 和 Flags，并将控制返回给 INT 之后的指令。

外部和内部中断。外部中断是由处理器外部的设备引起的。能传输外部中断信号的两条线路是非屏蔽中断(NMI)线和中断请求(INTR)线。NMI 线报告内存和 I/O 奇偶错。即使使用 CLI 清零中断标志试图屏蔽外部中断，处理器也总是响应这种中断。INTR 线报告来自外部设备的请求，也就是中断 05H~0FH，对应定时器、键盘、串行端口、固定磁盘、软盘驱动器和并行端口。

内部中断是由于执行一条 INT 指令或发生溢出的除操作、以单步方式执行或请求外部中

断（如磁盘 I/O）的结果。程序经常使用非屏蔽的内部中断来访问 BIOS 和 DOS 例程。

24.5 BIOS 中断

这一节包括 00H 到 1BH 的 BIOS 中断。其他没有包括在内的操作只能由 BIOS 执行。

INT 00H: 除数为 0。由试图除零的操作调用，显示一个信息，通常将系统挂起。

INT 01H: 单步。由 DEBUG 和其他调试器使用，可以在程序执行过程中单步调试。

INT 02H: 非屏蔽中断(NMI)。在硬件出现严重错误的情况下使用，如奇偶错，它总是能产生中断。使用 CLI 指令(清除中断标志)不影响中断产生的条件。

INT 03H: 断点。用于调试器来停止程序的执行。DEBUG 中的 Go 和 Proceed 命令在程序中设定的断点处设置这一中断。DEBUG 取消单步方式，并且允许程序正常执行一直到 INT 03H，随后 DEBUG 又重置成单步方式。

INT 04H: 溢出。可能由运算操作引起，通常并不采取什么动作。

INT 05H: 打印屏幕。打印视频显示区的内容。使用 INT 05H 从内部激活中断，使用 Ctrl+PrintScrn 在外部激活中断。操作可以被打断，并保存光标位置，没有寄存器受影响。在 BIOS 数据区的 50:00 地址包含这一操作的状态。

INT 08H: 系统定时器。调整系统时间和日期(如果必要)的硬件中断。可编程定时器芯片每 54.9254 毫秒产生一次中断，大约为一秒 18.2 次。

INT 09H: 键盘中断。是由按下或放开键盘上的一个键产生的，详见第 10 章。

INT 0BH, INT 0CH: 串行设备控制。分别控制 COM1 和 COM2 端口。

INT 0DH, INT 0FH: 并行设备控制。分别控制 LPT2 和 LPT1 端口。

INT 0EH: 软盘控制。用信号表示磁盘的动作，如完成一个 I/O 操作。

INT 10H: 视频显示。在 AH 中接收一个功能号，表示屏幕方式，设置光标、滚动和显示，详见第 9 章中的描述。

INT 11H: 确定设备。确定系统中的可选择设备，将 BIOS 中 40:10H 的值返回给 AX(在启动时，系统执行此操作并且在 40:10H 中保存 AX，详见前节“BIOS 数据区”)。

INT 12H: 确定内存大小。在 AX 中返回基本内存的大小，以连续的 KB 统计。

INT 13H: 磁盘 I/O。接收 AH 中的功能号，分别表示磁盘状态、读扇区、写扇区、检验、格式化和得到诊断，第 19 章包括这些内容。

INT 14H: 输入/输出通信。提供 I/O 位数据流(即每次一位)给 RS232 串行端口。DX 应该包含 RS232 端口的代码(0~3 分别对应 COM1、2、3 和 4)。通过 AH 寄存器确定功能号。

功能 00H: 初始化通信端口。在 AL 中设置下列参数，按位号表示：

位/秒	奇偶	终止位	字长
7~5	4~3	2	1~0
100=1200	00=无	0=1	10=7 位
101=2400	01=奇	1=2	11=8 位
110=4800	10=无		
111=9600	11=偶		

下面例子中，设置 COM1 每秒 1200 位，无奇偶校验，一位终止位和数据长度 8 位：

```
MOV AH, 00H           ; 请求初始化端 11
MOV AL, 11100011B     ; 参数
MOV DX, 00            ; COM1 串行端口
INT 14H               ; 调用中断服务
```

操作在 AX 中返回通信端口状态(详见功能 03H)。

功能 01H：传送字符。在 AL 中装入要传送的字符，DX 中为端口号。返回时，操作在 AH 中设置端口状态(参见功能 03H)。如果操作不能传送字节，也要设置 AH 的第 7 位，这一位一般是用来报告超时错误。在使用此服务之前执行功能 00H。

功能 02H：接收字符。在 DX 中装入端口号。操作从通信线上接收一个字符送到 AL。同时在 AH 中设置端口状态(参见功能 03)，表示错误的位为 7、4、3、2、1，因此在 AX 中的非零值表示一个输入错误。在使用此服务前执行功能 00H。

功能 03H：返回通信端口状态。在 DX 中装入端口号。操作在 AH 中返回线状态(从端口 03FDH)，AL 中返回调制解调器状态(从端口 03FEH)：

AH(线状态)	AL(调制解调器状态)
7 超时	7 接收线信号检测
6 传输移位寄存器空	6 环指示器
5 传输保持寄存器空	5 数据设置就绪
4 发现 Break 错	4 发送清零
3 组帧错	3 改变接收线信号检测
2 奇偶错	2 环检测器的变化
1 过载错	1 数据设置就绪的变化
0 数据就绪	0 清除发送的变化

其他 INT 14H 功能是 04H(扩展初始化)和 05H(扩展通信端口控制)。因为 INT 14H 限制在 9 600bps，适合于做传输数据的实验。一个快速但更复杂的方法是通过 INS 和 OUTS 操作直接发送到端口地址，更进一步的讨论超过了本书的范围。

INT 15H：系统服务。这个精心设计的操作在 AH 中提供了相当多的功能，包括：

21H	加电自检	88H	确定扩展内存大小
43H	读系统状态	89H	处理器转换为保护模式
84H	操纵杆支持	C2H	鼠标接口

INT 16H：键盘中断。在 AH 中接收基本键盘输入的功能号，参见第 10 章。

INT 17H：打印机输出。通过 BIOS 提供一些打印功能，参见第 20 章中所讨论的。

INT 18H：ROM 基本入口。如果系统没有包含系统引导程序的磁盘，启动时调用 BIOS。

INT 19H：引导程序加载器。如果一个装有操作系统程序的磁盘(包括软盘)设备是可用的，则将磁道 0，扇区 1 读入内存中引导程序的单元 7C00H，并将控制传递到这一单元。这一操作可以当作软件中断使用，它不清除屏幕，也不初始化 ROM BIOS 中的数据。

INT 1AH：读取和设置时间。根据 AH 中的功能码读或者设置日时钟：

00H=读系统定时器时钟。在 CX 中返回计数的高位部分，在 DX 中返回低位部分。如果自从上一次读取已经超过了 24 个小时，操作在 AL 中设置一个非零值。

01H=设置系统定时器时钟。在 CX 中装入计数的高位部分, DX 中装入低位部分。

02H~07H。这些功能为实时钟服务处理时间和日期。

要确定一个程序执行了多长时间,可以先设置时钟为零,然后在程序结束时读取时间。

INT 1BH: 键盘 Break 控制。当按下 Ctrl+Break 时,使 ROM BIOS 传递控制到它的中断地址,这里要设置一个标志。

24.6 INT 21H 服务例程

下面是与本书有关的 DOS INT 21H 服务例程,所需的功能码在 AH 中。

01H: 键盘输入有回显(见第 10 章)。

02H: 显示字符(见第 8 章)。

03H: 通信输入。从串行端口读一个字符到 AL,更多的是使用一个基本服务例程 BIOS INT 14H。

04H: 通信输出。DL 包含要传送的字符,更多使用的是 BIOS INT 14H。

05H: 打印机输出。打印单个字符:

```

MOV     AH, 05H      ; 请求打印
MOV     DL, char     ; 单个字符
INT     21H          ;

```

06H: 直接键盘输入和显示输出。这是一个很奇特的操作,它可以传送任何字符或者控制码。有两个版本,一个是有关键盘输入的,一个是有关屏幕输出的。对于输入,在 DL 中装入 0FFH。如果在键盘缓冲区里没有字符,操作设置 ZF,并且不等待输入。如果缓冲区内有字符,操作将字符装入 AL,并清零 ZF。此操作不在屏幕上回显字符,也不检查 Ctrl+Break 和 Ctrl+PrintScrn。AL 中的非 0 值表示是一个标准的 ASCII 字符,如字母或数字。AL 为 0 说明用户按下了一个扩展功能键,如 Home、F1 或 PageUp。为了得到 AL 中的扫描码,要立即重复调用 INT 21H 操作。对于屏幕输出,在 DL 中装入 ASCII 字符(不是 0FFH)。

07H: 直接键盘输入,不回显(见第 11 章)。

08H: 键盘输入,不回显(见第 10 章)。

09H: 显示字符串(见第 8 章)。

0AH: 有缓冲的键盘输入(见第 10 章)。

0BH: 检验键盘状态(见第 10 章)。

0CH: 清除键盘缓冲区并调用输入功能(见第 10 章)。

0DH: 重置磁盘驱动器(见第 18 章)。

0EH: 选择默认磁盘驱动器(见第 18 章)。

19H: 确定默认磁盘驱动器(见第 18 章)。

1AH: 设置磁盘传输区(DTA)(见第 18 章)。

1FH: 得到默认驱动器参数块(见第 18 章)。

25H: 设置中断向量。当用户按下 Ctrl+Break 或者 Ctrl+C 时,正常的过程是终止程序并返回操作系统。可以让程序提供自己的例程来处理这一情况,下面的例子使用功能 25H 来设

置它自己的 Ctrl+Break 处理程序 C10BREAK 在中断向量表中的地址(INT 23H), 这个例程可以采取任何必要的动作。

```

MOV     AH, 25H           ; 请求设置
MOV     AL, 23H           ; INT 23H 的表地址
LEA     DX, C10BREAK      ; 新地址
INT     21H               ; 调用中断服务
--
C10BREAK:                  ; Ctrl+Break 处理程序
--
IRET                    ; 中断返回

```

29H: 解析文件名(见第 18 章)。

2AH: 获取系统日期。返回下列二进制值: AL=星期(星期日=0); CX=年(1980~2099); DH=月(01~12); DL=日(01~31)。

2BH: 设置系统日期。加载下列二进制值: CX=年(1980~2099); DH=月(01~12); DL=日(01~31)。返回时, AL 指示有效操作(00H)或无效操作(FFH)。

2CH: 获取系统时间。返回下列二进制值: CH=小时, 24 小时的形式(00~23, 午夜为 00); CL=分(00~59); DH=秒(00~59); DL=百分之一秒(00~99)。

2DH: 设置系统时间。加载下列二进制值: CH=小时, 24 小时的形式(00~23, 午夜为 00); CL=分(00~59); DH=秒(00~59); DL=百分之一秒(00~99)。返回时, AL 指示有效操作(00H)或无效操作(FFH)。

2EH: 设置/重置磁盘验证(见第 18 章)。

2FH: 获取当前磁盘传输区的地址(DTA)(见第 18 章)。

31H: 终止但保持驻留(见第 23 章)。

32H: 获取磁盘参数块(DPB)(见第 18 章)。

3300H: 获得 Ctrl+C 状态。如果 Ctrl+C 标志为关闭(0), 导致系统只在处理字符 I/O 功能 01H~0CH 时检验 Ctrl+C。如果标志为开启(1), 系统在处理其他功能时检验 Ctrl+C。为了得到状态, 在 AL 中设置子功能 00H。在 DL 中的返回值为 00H=不能检验, 01H=能检验。

3301H: 设置 Ctrl+C 状态。如果 Ctrl+C 标志为关闭(0), 导致系统只在处理字符 I/O 功能 01H~0CH 时检查 Ctrl+C。如果标志为开启(1), 系统在处理其他功能时检验 Ctrl+C。为了设置状态, 在 AL 中设置子功能 01H, 在 DL 中装入状态为 00H=设置检验关闭, 01H=设置检验开启。

3305H: 获得启动驱动器。在 DL 中返回用来加载系统文件的驱动器号(1=A, 等等)。

3306H: 获取 DOS 信息。返回如下值:

BL=主版本号, 如版本 n.11 中的 n。

BH=次级版本号, 如版本 n.11 的 BH(11)。

DL=位 2~0 为修订号。

DH=DOS 版本标志(指示系统是否在常规存储器, 高地址存储器区或 ROM 中运行)。尽管命令 SETVER 可以设置版本号, 但功能 3306H 传送的是真实的版本号。

35H: 获取中断向量表的地址(见第 23 章)。

36H: 获取空闲磁盘空间(见第 18 章)。

38H: 获取/设置相关国家的信息。支持关于各个国家的信息的功能,如国家货币的符号和版式,千位和小数位的分隔符,日期和时间分隔符。在 DX 装入操作码:FFFFH 为设置系统一直使用的国家码,直到进一步的说明,或者装入其他当前使用的国家代码。

39H: 创建子目录(MKDIR)(见第 18 章)。

3AH: 删除子目录(RMDIR)(见第 18 章)。

3BH: 改变当前目录(CHDIR)(见第 18 章)。

3CH: 创建文件(见第 17 章)。

3DH: 打开文件(见第 17 章)。

3EH: 关闭文件(见第 17 章)。

3FH: 读文件/设备(见第 17 章)。

40H: 使用文件代号写文件/设备(见第 8、17 和 20 章。)

41H: 从目录中删除文件(见第 18 章)。

42H: 移动文件指针(见第 17 章)。

43H: 检查/改变文件属性(见第 18 章)。

44H: 设备的 I/O 控制。支持检查设备和读写数据的扩展子功能集(见第 23 章)。

45H: 复制文件代号(见第 18 章)。

46H: 强制复制文件代号(见第 18 章)。

47H: 获取当前目录(见第 18 章)。

48H: 分配内存块(见第 23 章)。

49H: 自由分配内存块(见第 23 章)。

4AH: 设置分配的内存块大小(见第 23 章)。

4BH: 加载/执行一个程序(见第 23 章)。

4CH: 终止程序。程序执行结束的标准操作(见第 4 章)。

4DH: 取子程序回送值(见第 23 章)。

4EH: 寻找到第一个匹配的目录入口项(见第 18 章)。

4FH: 寻找到下一个匹配的目录入口项(见第 18 章)。

50H: 设置程序段前缀地址(PSP)。在 BX 中加载当前程序 PSP 地址的偏移量,没有返回值。

51H: 获得程序段前缀地址(PSP)。返回当前程序 PSP 地址的偏移量(见第 23 章)。

54H: 获得检验状态(见第 18 章)。

56H: 重命名文件(见第 18 章)。

57H: 获取/设置文件日期和时间(见第 18 章)。

5800H: 获得内存分配策略(见第 24 章)。

5801H: 设置内存分配策略(见第 24 章)。

5802H: 取得上部存储器连接(见第 23 章)。

5803H: 设置上部存储器连接(见第 23 章)。

59H: 获取扩展错误码(见第 18 章)。

5AH: 创建临时文件(见第 18 章)。

5BH: 创建新文件(见第 18 章)。

5DH: 设置扩展错误。在 DX 中加载错误信息表的偏移地址。紧接着执行功能 59H(获得扩展错误码)重新得到表地址(见第 23 章)。

62H: 获得 PSP 地址。(和功能 51H 相同的操作。)

65H: 获得扩展国家信息。支持一系列关于各个国家信息的子功能。

67H: 设置最大文件代号计数(见第 23 章)。

68H: 提交文件(见第 18 章)。

6CH: 扩展打开文件。和功能 3CH(创建文件), 3DH(打开文件)和 5BH(创建不重名文件)一起使用(见第 18 章)。

24.7 端 口

端口是连接处理器和外部世界的设备。通过端口, 处理器可以从输入设备接收信号, 也可以输出信号到输出设备。端口是根据其地址来识别的, 端口地址的范围为 0H ~ 3FFH, 共有 1024 个端口。要注意的是, 这些地址不是传统的存储器地址。

下面列出了一些主要的端口地址:

020H~021H 可编程中断控制器端口。对外部中断做出响应, 如键盘、磁盘驱动器和系统时钟等外部设备。来自这些设备的请求中断处理器的当前工作并要求它按请求动作。这些端口是:

20H 8259 端口地址, 当操作发送 20H 到端口时, 它发出中断结束(EOI)信号。

21H 8259 中断屏蔽寄存器, 表明中断允许(0)或者中断禁止(1)。各位和 IRQ 表示的设备相同:

IRQ 0 系统定时器	IRQ 4 串行端口(COM1)
1 键盘	5 并行端口(LPT2)
2 二级 I/O 通道	6 软盘控制器
3 串行端口(COM2)	7 并行端口(LPT1)

40H~42H 8253 可编程时间间隔定时器端口。包含 3 个端口, 其处理功能如下:

40H 通道 0 计数寄存器, 每秒中断系统 18.2 次(每 55 毫秒计数 1 次, 计算值为 0~65535)并更新系统时钟。

41H 通道 1, 中断直接存储器访问(DMA)控制器以刷新内存(RAM 芯片保留数据只有几毫秒)。

42H 通道 2, 控制与端口 61H 相连的扬声器。

060H 键盘。处理键盘扫描码。

061H 8255 接口通道的端口, 下列位为 0 值时有如下处理功能:

- 位 0 8253 时间间隔定时器(端口 42H)时钟无效
- 位 1 扬声器无效
- 位 4 RAM 奇偶错有效
- 位 6 键盘声音关闭
- 位 7 键盘有效

200H~20FH 游戏控制器
 278H~27FH 并行打印机端口(LPT3)
 2F8H~2FFH 串行端口(COM2)
 378H~37FH 并行打印机端口(LPT2)
 3B0H~3BBH 单色显示端口
 3BCH~3BFH 并行打印机端口(LPT1)
 3C0H~3CFH VGA 端口
 3DAH~3DFH VGA 彩色 CRT 状态寄存器(只读)
 3F0H~3F7H 磁盘控制器
 3F8H~3FFH 串行端口(COM1)

尽管标准输入输出操作使用 INT 操作, 但访问端口 21H、40~42H、60H、61H 和 201H 时, 避开 BIOS 是安全的。例如, 引导 ROM BIOS 程序时, 扫描系统的串行和并行端口地址。如果找到端口地址, BIOS 将串行端口地址放在从内存单元 40:00H 开始的数据区, 并行端口的地址放在从单元 40:08H 开始的数据区。每个区有 4 个字入口项。BIOS 系统表有 2 个串行端口和 2 个并行端口, 如下所示:

40:00	F803	COM1
40:02	F802	COM2
40:04	0000	未用
40:06	0000	未用
40:08	7803	LPT1
40:0A	7802	LPT2
40:0C	0000	未用
40:0E	0000	未用

例如, 使用 BIOS INT 17H 打印一个字符, 在 DX 中插入打印机端口号:

```

MOV AH, 00H          ; 请求打印
MOV AL, char          ; 打印的字符
MOV DX, 0             ; 打印机端口 0=LPT1
INT 17H               ; 调用中断服务
  
```

一些程序只允许使用 LPT1 打印。如果系统装有 LPT1 和 LPT2 两个打印端口, 通过下面的程序可以交换(反转触发)它们在 BIOS 表中的地址。BIOSDATA 定义 BIOS 数据区, PARLPORT 定义第一个 4 字大小的端口地址。

```

BIOSDATA SEGMENT AT 40H          ; BIOS 数据区
        ORG      8H              ; 打印机端口地址
PARLPORT DW      4 DUP(?)        ; 4 个字
BIOSDATA ENDS

-
ASSUME DS: BIOSDATA
MOV     AX, BIOSDATA             ; BIOS 数据区的
MOV     DS, AX                   ; 初始地址在 DS 中
MOV     AX, PARLPORT(0)          ; LPT1 地址给 AX
MOV     BX, PARLPORT(2)          ; LPT2 地址给 BX
MOV     PARLPORT(0), BX          ; 交换
  
```

```
MOV     PARLPORT(2), AX    ; 地址
...
```

24.7.1 IN 和 OUT 指令

IN 和 OUT 指令可以在端口级直接处理 I/O。IN 从一个输入端口传递数据给 AL(字节)或 AX(字), 而 OUT 将 AL(字节)或 AX(字)中的数据传递到一个输出端口。格式如下:

[标号:]	IN	accum-reg, 端口
[标号:]	OUT	端口, accum-reg

可以静态或动态地指定端口地址。

静态。直接使用 0~255 的一个操作数:

```
输入      IN    AL, port#    ; 从端口输入一个字节
输出      OUT   port#, AX    ; 输出一个字到端口
```

动态。间接使用 DX 中 0~65535 的内容, 可以使用这个方法通过 DX 加 1 连续地处理端口地址。使用端口 60H 的例子如下:

```
MOV      DX, 60H          ; 端口 60H(键盘)
IN       AL, DX           ; 从端口得到字符
```

24.7.2 随机数发生器

下面的程序段可以利用 8253 时间间隔定时器产生随机数:

```
MOV AX, 0                ; 间隔定时器
OUT 43H, AL              ; 通过端口 43H
IN AL, 40H               ; 对端口 40H
IN AL, 40H               ; 访问 2 次
```

现在随机数应当在 AL 中。

24.8 串输入/输出

INSn 和 OUTSn 串指令也可以传输数据, 它们的工作很像第 11 章介绍的串指令:

INSn 指令。INSn 指令的操作如下:

指令	示例
INS(80286+)	INS ES:destination, DX
INSB(80286+)	REP INSB
INSW(80286+)	REP INSW
INSD(80386+)	REP INSD

接收的数据(或目的文件)是通过 ES: DI 寻址的“串”, DX 中包含输入端口的地址。通

常使用 **INSn** 要带 **REP** 前缀, **CX** 包含要接收的项(字节、字或双字)数。如果方向标志(**DF**)为 0, **DI** 根据每个接收到的项的大小增量, 如果 **DF** 为 1, **DI** 减量。

下例说明 **INSn** 操作:

```
MOV    CX, no_bytes      ; 字节数
LFA    DI, destination   ; 目的串(ES:DI)
MOV    DX, port-no       ; 从端口
REP    INSB              ; 接收字节
```

OUTSn 指令. **OUTSn** 指令的操作如下:

指令	示例
OUTS(80286+)	OUTS DX, DS:source
OUTSB(80286+)	REP OUTSB
OUTSW(80286+)	REP OUTSW
OUTSD(80386+)	REP OUTSD

发送的数据(或源)是通过 **DS:SI** 寻址的串, **DX** 中包含输出端口的地址。通常使用带有 **REP** 前缀的 **OUTSn** 指令, **CX** 中含有要发送的项(字节、字或双字)数。如果方向标志(**DF**)为 0, **SI** 根据每个接收到的项的大小增量, 如果 **DF** 为 1, **SI** 减量。

下例说明 **OUTSn** 操作:

```
MOV    CX, no_bytes      ; 字节数
LEA    SI, source        ; 源串(DS: SI)
MOV    DX, port-no       ; 从端口
REP    OUTSB             ; 发送字节
```

24.9 产生声音

PC 机通过内置的永久磁铁扬声器产生声音, 扬声器和端口 42H、43H 和 61H 连接。扬声器发声的操作步骤如下:

1. 得到端口 61H 的状态, 并保存。
2. 为了打开扬声器, 发送在位 0 和 1 中的位串 11 到端口 61H。端口激活 Intel 8255 可编程外围接口(PPI)芯片。
3. 为了关闭扬声器, 发送在位 0 和 1 中的位串 00 到端口 61H。

图 24-1 中的部分程序按照升序产生一系列音符。**DURATION** 提供了每个音符的长度, **TONE** 确定了频率。程序开始访问端口 61H, 并保存操作传送的值。时间间隔定时器产生时钟“嘀答”, 每秒 18.2 次“嘀答”, 每次“嘀答”中断程序的执行, 并引起振动发声。

TONE 中的内容决定了声音的频率, 高值导致低频率, 低值导致高频率。程序发出每一个音符后, 通过右移一位(相当于除 2)来增加 **TONE** 的频率。因为在此例中降低 **TONE** 将减少发音的长度, 程序同时左移一位(相当于乘 2)来增加 **DURATION**。

当 **TONE** 降低为零时, 程序结束。**DURATION** 和 **TONE** 中的初始值没有技术上的含义。可以实验使用其他的值, 并执行没有 **CLI** 指令的程序。

可以使用任何逻辑的变化来演奏一串音符序列，其目的可能是为了引起用户的注意。

		.DATA		
	DURATION	DW	10000	; 音符长度
	TONE	DW	512H	; 频率
				;
		.CODE		
		...		
		IN	AL, 61H	; 得到端口状态
		PUSH	AX	; 并保存
A20:		MOV	DX, DURATION	; 设置持续时间
A30:				
		OR	AL, 00000011B	; 设置各位为 0 或 1
		OUT	61H, AL	; 传递给扬声器
		MOV	CX, TONE	; 设置长度
A40:				
		LOOP	A40	; 时间延迟
		OR	AL, 00000010B	; 设置位 1 为打开状态
		OUT	61H, AL	; 传递给扬声器
		MOV	CX, TONE	; 设置长度
A50:				
		LOOP	A50	; 时间延迟
		DEC	DX	; 减少持续时间
		JNZ	A30	; 继续?
		SHL	DURATION, 1	; 否, 增加长度
		SHR	TONE, 1	; 减少频率
		JNZ	A20	; 为 0?
		POP	AX	; 重置
		AND	AL, 11111100B	; 端口
		OUT	61H, AL	; 值
		...		

图 24-1 产生声音

24.10 要 点

- ROM 驻留在 FFFF0H 开始的位置。接通电源使处理器进入重置状态，设置所有存储器单元为 0，对存储器进行奇偶校验，设置 CS 寄存器为 FFFF[0]H 和 IP 为 0。因此要执行的第一条指令在 FFFF:0，即 FFFF0H，它指向 BIOS 的入口。
- 引导时，BIOS 检查各个端口以识别和初始化所连接的设备。然后 BIOS 从存储器位置 0 开始建立中断向量表，它包含产生中断的地址。BIOS 执行的两个操作是确定设备和存储器的大小。BIOS 访问包含引导程序加载器的第一个磁盘扇区。
- BIOS 在段地址 40[0]H 开始的低地址存储器中保留自己的数据区。有关的数据区包括串行端口、并行端口、系统设备、键盘、磁盘驱动器、视频控制器、硬盘和实时钟。
- 中断指令如 INT 21H，它的操作数表明请求的类型。对 256 种可能类型的每一种，系统都在地址 0000H ~ 3FFH 的中断向量表中保留一个 4 字节地址。
- BIOS 中断类型的范围从 00H 到 1FH，包括打印屏幕、定时器、视频控制、软盘控制、视频显示、确定设备和内存大小、磁盘 I/O、键盘输入、通信、打印机输出和引导程序加载器。
- DOS INT 21H 处理的操作如磁盘输入、显示输出、打印机输出、重置磁盘、打开/关

闭文件、删除文件、读写记录、终止但保持驻留、创建子目录和终止程序。

- 通过端口，处理器从输入设备接收信号或发送信号到输出设备。端口通过它们的地址来识别，端口地址在 00H~3FFH 范围内，共有 1 024 个端口。
- PC 机通过内置永久磁铁扬声器产生声音，扬声器和端口 42H、43H 和 61H 相连。

24.11 习 题

24-1. 区别内部中断和外部中断。

24-2. 区别 NMI 连线和 INTR 连线。

24-3. (a) 指向 BIOS 入口的存储器地址是什么？(b) 启动时，系统如何定向到这一地址？

24-4. 引导时，BIOS 执行 INT 11H、12H 和 19H 操作。解释每个中断的用途。

24-5. BIOS 数据区的开始位置在哪里？

24-6. 下列二进制值记录在 BIOS 数据区。对每一项，确定是什么字段，并解释为 1 的位的意义。

(a) 10-11H: 01000100 01100111 (b) 17H: 11101010

(c) 18H: 00000001 (d) 96H: 00011010

24-7. 下列十六进制值记录在 BIOS 数据区。确定每一项是什么字段，并解释这些值的意义。

(a) 00-03H: F8 03 F8 02 (b) 08-0BH: 78 03 00 00

(c) 13-14H: 80 02 (d) 15-16H: 00 10

(e) 4AH-4BH: 50 00 (f) 60-61H: 0E 0D

(g) 84H: 18

24-8. 识别下列 BIOS INT 操作：(a) 单步模式，(b) 通信 I/O，(c) 获得设备状态，(d) 打印屏幕，(e) 磁盘 I/O，(f) 键盘输入，(g) 键盘，(h) 视频显示，(i) 打印机输出，(j) 读取和设置时间。

24-9. 确定下列 INT 21H 服务的功能：(a) 终止但保持驻留，(b) 获得中断向量表地址，(c) 创建子目录，(d) 获得空闲磁盘空间，(e) 获得 PSP 地址，(f) 创建新文件，(g) 获得系统时间，(h) 重命名文件。

24-10. 识别下列 INT 21H 的功能：(a) 05H，(b) 09H，(c) 0DH，(d) 19H，(e) 2BH，(f) 31H，(g) 36H，(h) 3AH，(i) 42H。

24-11. 参考 24.7 节中交换 LPT1 和 LPT2 地址的程序并修改指令，交换 COM1 和 COM2 的地址。

24-12. 针对下列要求修改图 24-1 程序：产生频率下降的音符；初始化 TONE=01 和 DURATION 为一个大值。在每次循环中，增加 TONE 的值，减少 DURATION 的值，当 DURATION 等于 0 时结束程序。

操作符与伪操作

目的：提供汇编语言操作符与伪操作的详细说明

25.1 引言

各种各样的汇编语言特性最初会让人感到有点不知所措。但是，一旦对前些年所描述的比较简单而通用的特性越来越熟悉，你会发现在这一章里说明的各种类型的区分符、操作符和伪操作更加容易理解和引用。汇编语言手册还包括了其他一些专门的特性。

注意，标准的 Turbo 汇编程序——TASM 可以在 MASM 方式下运行，即它接受标准 MASM 的规范；也可以在理想方式下运行，此时，在许多情况下，它使用的术语与规则会有所不同，并且有可能无法识别 MASM 的规范。

25.2 类型区分符

类型区分符可以指示数据变量的大小或指令标号的相对距离。标明数据变量大小的区分符是 BYTE, WORD, DWORD, FWORD, QWORD, 以及 TBYTE。标明指令标号距离的区分符是 NEAR, FAR, 以及 PROC。近地址仅是简单的偏移地址，用于当前段之内；远地址是由段：偏移地址组成的，可以访问另一段中的数据。

PTR 与 THIS 操作符以及 COM、EXTRN、LABEL 和 PROC 伪操作使用类型区分符。

25.3 操作符

操作符为在汇编期间修改与分析操作数提供了方便。操作符分为以下各种类型：

- 计算操作符：算术运算，变址，逻辑，移位，以及结构字段名。
- 宏操作符：在第 21 章里所涉及的各种类型。
- 记录操作符：MASK 和 WIDTH，在本章后面的 RECORD 伪操作部分会涉及到它们。
- 关系操作符：EQ, GE, GT, LE, LT, 以及 NE。
- 段操作符：OFFSET, SEG, 以及段跨越。

• **类型(或属性)操作符:** HIGH, HIGHWORD, LENGTH, LOW, LOWWORD, PTR, SHORT, SIZE, THIS, 以及 TYPE。

由于有关这些类型方面的知识并不是必需的, 所以本章只是按操作符字母顺序简单地加以介绍。

(1) **算术运算操作符。**该操作符包括常见的一些算术符号和在汇编期间实现的算术运算。尽管使用这些操作符的优点是每次修改程序和重新汇编时, 汇编程序会自动地重新计算这些算术操作符的值, 但在大多数情况下, 可以自己完成这些计算。下面是这些操作符的表, 同时还提供了使用它们的例子以及得到的结果:

符号类型	例	结果
+ 加法	FLDA+25	把 25 加到 FLDA 的地址上
+ 正	+FLDA	把 FLDA 作为正数处理
- 减法	FLDB-FLDA	计算 2 个偏移地址间的差
- 负	-FLDA	FLDA 的符号变反
* 乘法	值 * 3	值乘以 3
/ 除法	值/3	值除以 3
MOD 余数	值 1 MOD 值 2	值 1/值 2 得到的余数

除加法(+)和减法(-)外, 所有操作符必须是整数常数。以下相关的例子说明整数表达式:

```
value=12*4      ; 48
value=value/6    ; 48/6=8
value=-value-3   ; (-8)-(3)=-11
```

算术操作符常用于相等伪操作, 在“伪操作”一节会涉及到它。

(2) **HIGH 与 HIGHWORD 操作符。**HIGH 操作符回送表达式的高位(最左边)字节, 而 HIGHWORD(MASM 6.0 以后)则回送表达式的高位字(也可参见 LOW 操作符)。举例如下:

```
EQUVAL    EQU    1234H
...
MOV  CL, HIGH EQUVAL      ; 把 12H 装入 CL
```

(3) **变址操作符。**对于存储器间接寻址, 操作数访问基址或变址寄存器、常数、偏移变量以及变量。变址操作符使用方括号, 相当于加(+)符号的作用。变址的典型用法是访问表中的数据项。可以用以下操作访问变址存储器:

• 把在方括号中的立即数或名字, 编码成为[常数]。例如, 把 PART_TBL 的第 5 项装入 CL(注意, PART_TBL[0]是第 1 项):

```
PART_TBL  DB    25  DUP(?)      ; 定义表格
...
MOV  CL, PART_TBL[4]           ; 取得第 5 项
```

• 把基址寄存器 BX 作为[BX]和段寄存器 DS 联合在一起, 把基址寄存器 BP 作为[BP]和段寄存器 SS 联合在一起。例如, 使用 BX(作为 DS: BX)中的偏移地址, 把所访问的项传送到 DX:

```
MOV  DX, [BX]                ; 基址寄存器 DS: BX
```

- 把变址寄存器 DI 作为[DI]并把变址寄存器 SI 作为[SI], 将它们和 DS 段寄存器联合在一起。例如, 使用 SI(作为 DS: SI)中的偏移地址, 把所访问的项传送到 AX:

```
MOV AX, [SI] ; 变址寄存器 DS: SI
```

- 组合变址寄存器。例如, 把 AX 的内容传送到由 DS 的地址、BX 的偏移地址、SI 的偏移地址和常数 4 相加所确定的地址中:

```
MOV [BX+SI+4], AX ; 基址+变址+常数
```

上面例子中的第一个操作数也可以写成[BX+SI]+4。可以按任何次序组合这些操作数, 但不能组合 2 个基址寄存器[BX+BP]或 2 个变址寄存器[DI+SI]。变址寄存器必须在方括号内, 这使得汇编程序知道按变址操作符处理它。

(4) **LENGTH 操作符**。LENGTH 操作符回送由 DUP 操作符定义的项目数, 如下面的 MOV 指令所示:

```
PART_TBL DW 10 DUF(?)
...
MOV DX, LENGTH PART_TBL ; 把长度 10 回送到 DX
```

如果所访问的操作数不包含 DUP 项目, 那么该操作符回送值 01(对其可用性的限制)(也可参见 SIZE 和 TYPE 操作符)。

(5) **逻辑操作符**。逻辑操作符处理表达式中的位:

操作符	用于	效果
AND	expression1 AND expression2	各位相与
OR	expression1 OR expression2	各位相或
XOR	expression1 XOR expression2	各位异或
NOT	NOT expression1	各位求反

下面是 2 个例子:

```
MOV CL, 00111100B AND 01010101B ; CL=00010100B
MOV DL, NOT 01010101B ; DL=10101010B
```

(6) **LOW/LOWWORD 操作符**。LOW 操作符回送表达式的低位(最右边)字节, 而 LOWWORD(MASM 6.0 以后)回送表达式的低位字 (也可参见 HIGH 操作符)。下面是一个例子:

```
EQU_VAL EQU 1234H
...
MOV CL, LOW EQU_VAL ; 把 34H 装入 CL
```

(7) **OFFSET 操作符**。OFFSET 操作符回送变量或标号的偏移地址。该操作符编码为 OFFSET variable/label。下面的 MOV 回送 PART_TBL 的偏移地址:

```
MOV DX, OFFSET PART_TBL
```

注意, 指令 LEA DX, PART_TBL 回送同样的值但不需要 OFFSET。

(8) **MASK 操作符**。见下一节“伪操作”中的“RECORD 伪操作”。

(9) **PTR 操作符**。PTR 操作符可以用在数据变量和指令标号中。它使用类型区分符 BYTE、WORD、FWORD、DWORD、QWORD 和 TBYTE 指定具有二义性的操作数大小，或指定变量所定义类型(DB、DW、DF、DD、DQ 或 DT)之外的大小。还使用类型区分符 NEAR、FAR 和 PROC 指明标号距离。

该操作符编码为 type PTR expression，其中 type 是个新属性，如 BYTE，而 expression 则是变量或常数。下面是 PTR 操作符的例子：

```

BYTEA    DB      22H
          DB      35H
WORDA    DW      2672H           ; 数据按 7226H 存放
...
ADD      BL, BYTE PTR WORDA+1   ; 加第二个字节(26)
MOV      BYTE PTR WORDA, 05     ; 把 05 传送给第一个字节
MOV      AX, WORD PTR BYTEA     ; 把 2 个字节(2235)传送到 AX
CALL     FAR PTR [BX]           ; 调用远过程

```

与 PTR 实现同样功能的是 LABEL 伪操作，稍后再加以讨论。

(10) **SEG 操作符**。SEG 操作符返回指定的变量或标号所在段的段地址。把分别汇编的段组合起来的程序很可能会使用这个操作符。该操作符编码为 SEG variable/label。

以下各不相关的 MOV 语句回送引用的名字所在段的段地址：

```

MOV  AX, SEG WORDA           ; 取数据段地址
MOV  AX, SEG A10BEGIN        ; 取代码段地址

```

(11) **段跨越操作符**。该操作符编码为冒号形式(:)，它计算相对于一个特殊段的标号或变量的地址。该操作符编码为 segment: expression。段可以是段寄存器，段，或组名中的任何一个。表达式可以是常数、表达式或 SEG 表达式。

以下这些例子跨越默认的 DS 段寄存器：

```

MOV  BH, ES: 10H             ; 从 ES+10H 处取数
MOV  CX, SS: [BX]            ; 从 SS+BX 中的偏移值处取数

```

段跨越操作符适用于只有一个操作数的情况。

(12) **SHL 与 SHR 操作符**。操作符 SHL 和 SHR 在汇编期间使表达式移位。该操作符编码成 expression SHL/SHR count。在下面的例子中，SHR 操作符将位常数向右移 3 位：

```

MOV  BL, 01011101B SHR 3     ; 装入 00001011B

```

表达式很有可能引用符号名而不是常数值。

(13) **SHORT 操作符**。SHORT 操作符的用途是修改 JMP 在+127~-128 字节目的地范围内的 NEAR 属性，编码为 JMP SHORT label。汇编程序把机器码操作数从两个字节减少为一个字节。这一特性对于向前转移的近跳转是有用的，因为不然的话，汇编程序最初不知道跳转地址的距离，可能会为近跳转采用 2 个字节。

(14) **SIZE 操作符**。SIZE 操作符回送 LENGTH 与 TYPE 的乘积，并且仅当所引用的变量包含 DUP 项时才是有用的。该操作符编码为 SIZE variable。相关例子见“TYPE 操作符”。

(15) **THIS 操作符**。THIS 操作符建立一个具有段与偏移值的操作数，该操作数的段和

偏移值等于当前单元计数器的相应值。该操作符编码成: **THIS type**, **type**(类型)对于变量可以是 **BYTE**、**WORD**、**DWORD**、**FWORD**、**QWORD** 或 **TBYTE**, 对于标号可以是 **NEAR**、**FAR** 或 **PROC**。

THIS 典型地可以和 **EQU** 或相等 (=) 伪操作一起使用。以下例子定义 **PART_REC**:

```
PART_REC EQU THIS BYTE
```

效果和使用 **LABEL** 伪操作是一样的, 如

```
PART_REC LABEL BYTE
```

(16) TYPE 操作符。**TYPE** 操作符根据所引用变量的定义回送字节数。但是, 该操作对于串变量总是回送 1, 对于常数则回送 0。该操作符的编码为 **TYPE variable/label**。

定义	数字变量的字节数
DB/BYTE	1
DW/WORD	2
DD/DWORD	4
DF/FWORD	6
DQ/QWORD	8
DT/TWORD	10
STRUC/STRUCT	由结构定义的字节数
NEAR label	FFFFH
FAR label	FFFEH

以下例子说明 **TYPE**, **LENGTH** 和 **SIZE** 操作符:

```

BYTEA    DB    ?                ; 定义 1 个字节
PART_TBL DW    10 DUP(?)        ; 定义 10 个字节
...
MOV AX, TYPE BYTEA              ; AX=0001H
MOV AX, TYPE PART_TBL           ; AX=0002H
MOV CX, LENGTH PART_TBL         ; CX=000AH(10)
MOV DX, SIZE PART_TBL           ; DX=0014H(20)

```

因为 **PART_TBL** 定义为 **DW**, **TYPE** 回送 **0002H**, **LENGTH** 在 **DUP** 项的基础上回送 **000AH(10)**, 而 **SIZE** 回送类型乘以长度, 即 **14H(20)**。

(17) WIDTH 操作符。见下一节的“**RECORD** 伪操作”。

25.4 伪 操 作

本节讨论大多数汇编程序伪操作。第 4 章详细介绍了定义数据(**DB**, **DW**, 等等)的伪操作, 第 21 章介绍了宏指令的伪操作, 这里不再重复。伪操作分为以下各种类型:

- 代码标号: **ALIGN**, **EVEN**, **LABEL**, 以及 **PROC**。
- 条件汇编: **IF**, **ELSE** 及其他, 第 21 章已讨论过。
- 条件出错: **.ERR**, **.ERR1** 及其他。

- 数据分配: ALIGN, EQU, EVEN, LABEL, 以及 ORG (DB, DW, DD, DF, DQ, 以及 DT 在第 4 章已经介绍)。
- 列表控制: .CREF, .LIST, PAGE, SUBTTL, TITLE, .XCREF, 以及 .XLIST, 本章将会说明它们。(LALL, .LFCOND, .SALL, .SFCOND, .TFCOND, 以及 .XALL 在第 21 章讨论过。)
- 宏: ENDM, EXITM, LOCAL, MACRO, 以及 PURGE, 在第 21 章介绍过。
- 杂类: COMMENT, INCLUDE, INCLUDELIB, NAME, &OUT, 以及 .RADIX。
- 处理器: .8086, .286, .286P, .386, .386P, 等等。
- 重复块: IRP, IRPC, 以及 REPT, 在第 21 章讨论过。
- 作用域: COMM, EXTRN, 以及 PUBLIC。
- 段: .ALPHA, ASSUME, .DOSSEG, END, ENDS, GROUP, SEGMENT, 以及 .SEQ。
- 简化段: .CODE, .CONST, .DATA, .DATA?, .EXIT, .FARDATA, .FARDATA?, .MODEL, 以及 .STACK。
- 结构/记录: ENDS, RECORD, STRUCT, TYPEDEF, UNION。

以下各节按字母顺序介绍这些伪操作。

(1) **ALIGN(对准)伪操作**。ALIGN 伪操作使汇编程序把下一个数据项或指令按照给定值对准在一个地址上。对准可以为处理器访问字和双字提供方便。它的格式是 ALIGN number, 其中 number 必须是 2 的幂, 比如 2, 4, 8 或 16。

在下面的例子中, 单元计数器在 0005 处, ALIGN 4 语句使得汇编程序将单元计数器前进到下一个能被 4 整除的地址:

```
0005    ALIGN    4
0008    DBWORD  DD  0           ; 对准在双字边界上
```

如果单元计数器已经在所要求的地址处, 那么它就不动了。汇编程序对于数据把未用的字节用零填充而对于指令则用 NOP 填充。注意, ALIGN 2 和 EVEN 效果是一样的。

(2) **.ALPHA 伪操作**。该伪操作位于程序的起点或其附近, 它通知汇编程序为了和早期汇编程序版本兼容, 需要按字母顺序排列段。也可以在汇编程序命令行中使用 /A 选项。

(3) **ASSUME 伪操作**。ASSUME 通知汇编程序把段名与 CS、DS、ES 和 SS 段寄存器联系起来。它的格式是

ASSUME	segment-reg: segment-name [, ...]
--------	-----------------------------------

有效的 segment-reg 项是 CS, DS, ES, SS, FS, 以及 GS。有效的 segment-name 项是段寄存器的段名, NOTHING, GROUP, 以及 SEG 表达式。一个 ASSUME 语句可以按任何次序给多达 4 个段寄存器赋值。简化段伪操作自动生成 ASSUME。

在下面的 ASSUME 语句中, CODESEG、DATASEG 和 STACK 是程序用于定义段的名称:

```
ASSUME CS: CODESEG, DS: DATASEG, SS: STACK, ES: DATASEG
```

省略段的引用和编写代码 NOTHING 是一样的。关键字 NOTHING 的使用还可以取消以

前的任何 ASSUME 所指定的段寄存器如 ASSUME ES: NOTHING。既没有对 ES 赋值又没有使用 NOTHING 取消它, 那么对于引用数据段中的一个项, 指令操作数可以使用段跨越操作符(:) 引用 ES, 其中必须包含能正常执行的有效段地址:

```
MOV AX, ES: [BX]           ; 使用变址地址
MOV AX, ES: WORDA          ; 传送 WORDA 的内容
```

(4) **.CODE 伪操作**。这个简化段伪操作定义代码段。它的格式是 .CODE [name], 其中 name 是个可选项。所有可执行代码必须存放在这个段内。对于 Tiny, Small, 以及 Compact 存储模型, 默认的段名是 _TEXT。Medium, 和 Large 存储模型允许多个代码段, 可以根据 name 操作数来区分它们(参见.MODEL 伪操作)。

(5) **COMM 伪操作**。把变量定义成 COMM(公共)使它既有 PUBLIC 又有 EXTRN 属性。用这种方法, 不必把变量在一个模块中定义成 PUBLIC, 并且又在另一模块中定义成 EXTRN。COMM 伪操作的格式是

COMM	[NEAR/FAR]label: size[: count]
------	--------------------------------

- COMM 是在数据段内编码的。
- NEAR 或 FAR 属性可以被编码或默认为其中的一个, 这取决于存储模型。
- label 是变量的名字。注意, 该变量不能有初始值。
- size 可以是 BYTE、WORD、DWORD、QWORD 和 TBYTE 中的任何一种区分符, 或是一个指定字节数的整数。
- count 指明变量的元素数, 默认值是 1。

以下例子定义带 COMM 属性的项:

```
COMM NEAR COM_FLD1: WORD           ; 带 COMM 属性的字的大小
COMM FAR  COM_FDL2: BYTE: 25       ; 带 COMM 属性的 25 个字节
```

(6) **COMMENT 伪操作**。这个伪操作可以用于多行的注释。其格式为

COMMENT delimiter[comments] [comments] delimiter [comments]

定界符(delimiter)是第一个非空白字符, 比如 % 或 +, 跟在 COMMENT 后面。注释结束于第二个定界符出现的行上。这一例子使用 “+” 作为定界符:

```
COMMENT + This routine scans
         Keyboard input for
         + invalid characters
```

(7) **.CONST 伪操作**。这个简化段伪操作定义一个带 ‘const’ 类别的数据(或常数数据)段(参见.MODEL 伪操作)。

(8) **.CREF 伪操作**。这个伪操作(默认的)通知汇编程序产生交叉引用表。可以使用后面的.XCREF 伪操作禁止该表。

(9) **.DATA 和.DATA?伪操作**。这些简化段伪操作定义数据段。.DATA 定义一个初始化的近数据段; .DATA?则定义一个未初始化的近数据段, 通常在和高级语言连接时会使用它。对于一个独立的汇编程序, 也可以在.DATA 段内定义未初始化的近数据(参见.FARDATA

和.MODEL 伪操作)。

(10) **DOSSEG/DOSSEG** 伪操作。有许多方法控制汇编程序排列各段的序列。可以在程序的起点编写 .SEQ 或 .ALPHA 伪操作, 或者在汇编命令行上输入 /S 或 /A 选项。DOSSEG(MSAM 6.0 以后是 .DOSSEG)伪操作通知汇编程序忽略所有其他请求, 而去接受 DOS 段序列——基本上是代码、数据和堆栈。在程序的起点或附近编写这个伪操作, 主要是便于对独立的程序使用 CODEVIEW 调试程序。

(11) **END** 伪操作。END 伪操作放在源程序的末尾, 它的格式是 END [start-address]。选项 start-address 指明代码段开始执行的地址(通常是第一条指令)。系统装入程序使用这一地址初始化 CS 寄存器。如果程序只由一个模块组成, 那就定义一个起始地址; 如果是由许多模块组成的, 那么也只有一个模块(通常是第一个)有起始地址。

(12) **ENDP** 伪操作。ENDP 伪操作指明由 PROC 定义的过程的结束。它的格式是 procedure-name ENDP, 其中 procedure-name(过程名)与所定义的过程中的过程名是相同的。

(13) **ENDS** 伪操作。这个伪操作指明段(由 SEGMENT 定义的)的结束或是一个结构(由 STRUC 或 STRUCT 定义的)的结束。它的格式是: segment-name ENDS, 其中 segment-name 与所定义的段或结构中的名字是相同的。

(14) **相等伪操作**。相等伪操作的两种类型是等号和 EQU。它们都只在汇编期间进行处理而不进行任何存储。它们的用途举例来说是提供数字常数的名字和所定义项的别名。该伪操作应当在被引用前在程序中加以定义。

等号伪操作的格式是 name=expression, 其中 expression(表达式)是任意的整数值或数字表达式。该伪操作可以对一个项任意次赋值。以下是 2 个例子:

例 1: ROW=12

COL=16

MOV BX, ROW+COL

例 2: SCREEN_LOCS=25 * 0

MOV CX, SCREEN_LOCS

EQU 伪操作用于再定义数据名或具有其他数据名的变量, 变量, 或立即值。汇编程序用操作数来替代每次出现的名字。EQU 伪操作在程序中对一个项只能赋值一次。数字与串数据格式的区别是:

```
数字相等:      name EQU expression
串相等:        name EQU <string>
```

使用带数字数据 EQU 的例子是:

```
COUNTER      DW 0
SUM           EQU COUNTER      ; COUNTER 的另一个名字
TEN          EQU 10            ; 数字值
...
INC SUM       ; COUNTER 增量
ADD SUM, TEN  ; COUNTER 加 10
```

使用带串数据 EQU 的例子是:

```
PROD_MSG     EQU <'Enter product number: ' >
```

```

    BY_PTR      EQU  <BYTE PTR>
    ...
    MFSSGE1     DB  PROD_MSG           ; 用串替代
    ...
    MOV  SAVE, BY_PTR [BX]           ; 用串替代

```

尖括号(<>)使之更容易指明串操作数。

(15) **.ERR** 伪操作。下面这些条件出错伪操作有助于在汇编期间对错误的检测

伪操作	强制出错
.ERR	当遇到时
.ERR1	在汇编的第 1 遍扫描期间
.ERR2	在汇编的第 2 遍扫描期间
.ERRE	按真(0)表达式
.ERRNE	按假(非 0)表达式
.ERRDEF	按定义的符号
.ERRNDEF	按未定义的符号
.ERRB	按空白串
.ERRNB	按非空白串
.ERRIDN[I]	按相同的串
.ERRDIF[I]	按不同的串

自 MASM 6.0 以后, 已经不必关注汇编的第 1 遍扫描(.ERR1)和第 2 遍扫描(.ERR2)了。可以使用前面的宏与条件汇编语句中的伪操作。在下面的条件汇编语句中, 如果条件不为真, 则汇编程序将显示信息:

```

IF      condition
...
ELSE    .ERR
        %OUT [message]
ENDIF

```

(16) **EVEN** 伪操作。**EVEN** 通知汇编程序如果必要的话使其单元计数器向前进, 以便下一个定义的数据项或指令可以对准偶数存储边界。这一特性便于处理器一次存取 16 位或 32 位(**ALIGN 2** 产生与 **EVEN** 同样的效果)。

下列例子中, **BYTE_LOCN** 是偶数边界 0016 上的 1 字节字段, 现在单元计数器是 0017。**EVEN** 使汇编程序让单元计数器前进一个字节到 0018, 在那里, 下一个数据项 **WORD_LOCN** 被定义:

```

0016    BYTE_LOCN DB      ?
0017    EVEN                               ; 单元计数器向前进
0018    WORD_LOCN DW      ?

```

(17) **.EXIT** 伪操作。在代码段中, 可以使用 **.EXIT** 伪操作产生程序结束代码。它的格式是 **.EXIT [return-value]**, 其中 **return-value**(回送值)为 0 指没有问题, 而 1 的回送值指结束处理的错误。所产生的代码是:

```
MOV  AH, 4CH
```

```
MOV AL, return-value      ; 如果回送值被编码则产生
INT 21H
```

(18) **EXTRN/EXTERN 伪操作**。EXTRN(或 MASM 6.0 以后的 EXTERN)伪操作通知汇编程序和连接程序有关当前汇编所引用的数据变量和标号,它们是在另一个模块(与当前模块连接的)中定义的。伪操作的格式是 EXTRN name: type [, ...], 其中 name 是在另一个汇编模块中定义的项并在那里被声明为 PUBLIC 的。type 可以是下列的任何一种:

- 数据项: ABS(常数), BYTE, WORD, DWORD, FWORD, QWORD, TBYTE。在产生项的段中编写 EXTRN 代码。
- 距离: NEAR 或 FAR。在产生项的段中编写 NEAR 代码,可以在任何地方编写 FAR 代码。

在下面的例子中,调用程序把 CON_VALUE 定义为 PUBLIC 和 DW,被调用的子程序把 CON_VALUE(在另一个段中)标识为 EXTRN 和 FAR。

调用程序:

```
DSEG1 SEGMENT
        PUBLIC CON_VALUE
        ...
CON_VALUE DW ?
        ...
DSEG1 ENDS
```

被调用的子程序:

```
        EXTRN CON_VALUE: FAR
DSEG2 SEGMENT
        ...
MOV AX, WORD PTR CON_VALUE
        ...
DSEG2 ENDS
```

参见第 22 章有关 EXTRN 的例子。

(19) **.FARDATA 和 .FARDATA? 伪操作**。这些简化段伪操作定义数据段。.FARDATA 定义一个已经初始化的远数据段,而 .FARDATA? 则定义一个未初始化的远数据段。对于一个独立汇编的程序,也可以在 .FARDATA 段中定义未初始化的远数据(参见 .DATA 与 .MODEL 伪操作)。

(20) **GROUP 伪操作**。程序可以包含几个同样类型的段(代码,数据或堆栈)。GROUP 伪操作的用途是把同一类型的段集合在一个名字下,使它们常驻在一个段内,通常是在数据段内。格式是

name	GROUP	seg-name[, seg-name], ...
------	-------	---------------------------

下面的 GROUP 把 DATASEG1 和 DATASEG2 组合在同一个汇编模块中:

```
GROUPX GROUP DATASEG1, DATASEG2
DATASEG1 SEGMENT PARA 'Data'
        ASSUME DS: GROUPX
        ...
```

```

DATASEG1 ENDS
;
DATASEG2 SEGMENT PARA 'Data'
        ASSUME DS: GROUPX
...
DATASEG2 ENDS

```

使用 GROUP 的作用类似于给段以相同的名字和 PUBLIC 属性。

(21) INCLUDE 伪操作。如果有不同程序都能使用的一段汇编代码或宏指令，可以把它们存放在单独的磁盘文件中，以便提供给任何程序使用。考虑把 ASCII 码转换成二进制的例行程序，将其存放在名为 CONVERT.LIB 的文件中。为访问该文件，插入 INCLUDE 语句，如：

```
INCLUDE path: CONVERT.LIB
```

为把此语句插入源程序中，通常要写明 ASCII 转换例行程序的位置。汇编程序定位到磁盘上的这个文件并把这些语句包含在程序里(如果汇编程序没有找到该文件，会发出出错信息)。

对于每个所包含的行，汇编程序要在.LST 文件的第 30 列处打印一个 C(取决于版本)，而源代码是从第 33 列开始的。

第 21 章给出了 INCLUDE 的例子并说明了如何只针对汇编第 1 遍扫描使用该伪操作。

(22) LABEL 伪操作。LABEL 伪操作允许程序再定义指令标号或数据变量的属性。它的格式是 name LABEL type-specifier。对于标号，LABEL 允许把可执行代码再定义为 NEAR, FAR 或 PROC，比如进入过程的第二个入口点。对于变量，类型区分符(type-specifier)BYTE, WORD, DWORD, FWORD, QWORD 或 TBYTE，或者结构名，可以分别用来再定义数据项和结构名。例如，LABEL 允许程序把一个字段定义成既是 DB 又是 DW。而汇编程序不会使单元计数器前进。

以下例子说明 BYTE 和 WORD 类型，假定是从单元 010H 开始的。

```

010  BYTE1  LABEL  BYTE           ; 把第一个字节定义为 BYTE1,
010  WORD1  DW    2532H           ; 最先两个字节为 WORD1,
012  WORD2  LABEL  WORD           ; 第二与第四个字节为 WORD2,
012  BYTE2  DB    25H             ; 第三个字节为 BYTE2,
013                DB    32H       ; 第四个字节
...
      MOV    AX, BYTE1           ; 传送第一个字节
      MOV    BX, WORD2           ; 传送第二与第四个字节

```

第一条 MOV 指令只传送 WORD1 的第一个字节。第二条 MOV 指令传送从 BYTE2 开始的 2 个字节。PTR 操作符实现类似的功能。

(23) .LIST 伪操作。.LIST 伪操作(默认)使汇编程序列出源程序。可以有不需要列出来的代码块，因为它是和其他程序共用的。在这种情况下，可以使用 XLIST(或 NOLIST)伪操作中止列表，然后使用 .LIST 恢复列表。这些伪操作都不需要操作数。

(24).MODEL 伪操作。这个简化段伪操作产生默认段以及所要求的 ASSUME 与 GROUP 语句。它的格式是.MODEL memory-model，其中 memory-model(存储模型)如下所示：

- Tiny(微型)。代码与数据在一个段内用于 .COM 程序。
- Small(小型)。代码在一个段内($\leq 64\text{K}$)，数据在一个段内($\leq 64\text{K}$)。
- Medium(中型)。任意个代码段，数据在一个段内($\leq 64\text{K}$)。
- Compact(紧凑型)。代码在一个段内($\leq 64\text{K}$)，任意个数据段。
- Large(大型)。代码与数据均为任意个段，无 $>64\text{K}$ 数组。
- Huge(巨型)。代码与数据均为任意个段，数组可 $>64\text{K}$ 。
- Flat(平面型)。不分段，运行在保护模式下，用 32 位地址。

(25) **.NOLIST** 伪操作。(参见 XLIST 伪操作。)

(26) **ORG** 伪操作。考虑一个有以下定义的数据段：

偏移值	名字	操作	操作数	单元计数器
00	WORD1	DW	2542H	02
02	BYTE1	DB	36H	03
03	WORD2	DW	212EH	05
05	BYTE2	DD	00000705H	09

最初，汇编程序的单元计数器被设置为 00。由于 WORD1 是 2 个字节，单元计数器为定位下一个项要增加到 02。由于 BYTE1 是一个字节，单元计数器增加到 03，以此类推。可以使用 ORG 伪操作修改单元计数器的内容并相应地定位下一个定义的项。它的格式是 ORG expression，其中 expression 必须为 2 字节的绝对数而不能是符号名。

假设以下数据项是在上面定义的 BYTE2 后被立即定义的：

偏移值	名字	操作	操作数	单元计数器
		ORG	0	00
00	BYTE3	DB	?	01
01	WORD3	DW	?	02
03	BYTE4	DB	?	04
		ORG	\$+5	09

第一个 ORG 把单元计数器复位为 00。跟着的变量 BYTE3、WORD3 和 BYTE4 要把原来定义为 WORD1、BYTE1 和 WORD2 的存储单元分别重新定义。

最后的 ORG 中，操作数包含一个美元符(\$)，它表示单元计数器的当前值。因此，操作数 \$+5 把单元计数器置成 04+5，即 09，这和 BYTE2 定义之后的设置一样。

对 WORD2 的引用是位于偏移值 03 处的 1 字的字段，而对于 BYTE4 的引用则是在偏移值 03 处的 1 字节的字段：

```
MOV AX, WORD2      ; 一个字
MOV AL, BYTE4      ; 一个字节
```

当使用 ORG 再定义存储单元时，要保证单元计数器复位成正确的值，同时应把所有再定义的存储单元都考虑在内。再定义的变量不应当包含被定义的常数——这些常数会覆盖原来的常数。ORG 不能出现在 STRUC 定义内。

(27) **%OUT/ECHO** 伪操作。这个操作告诉汇编程序向标准输出设备(通常是屏幕)发出一条信息(message)(MASM 6.0 以后，名为 ECHO)。该伪操作的格式是 %OUT/ECHO message。“ERR 伪操作”一节给出了例子。

(28) **PAGE** 伪操作。PAGE 伪操作在源程序的起点指定汇编程序列在一页中的最大行数

和一行中的最大字符数。它的格式是 PAGE [[length], width]。例如, PAGE 60, 132 设置每页 60 行和每行 132 个字符。

每页的行数可以在 10 到 255 行范围内, 而每行的字符数可在 60 到 132 个范围内。省略 PAGE 语句汇编程序默认为 PAGE 50, 80。为了强制一页在指定行上退出来(比如在一个段的末尾), 编写不带操作数的 PAGE。

(29) PROC 伪操作。过程是由 PROC 伪操作开始而由 ENDP 结束的代码块。虽然从技术上说, 可以直接进入或者使用 JMP 指令进入一个过程, 但通常的做法是使用 CALL 进入并用 RETN 或 RETF 退出。CALL 的操作数可以是 NEAR 或 FAR 类型的区分符。

与调用过程在同一段的过程是 NEAR 过程, 它是用偏移值访问的, 格式是 procedure-name PROC [NEAR]。省略操作数默认为 NEAR。如果被调用的过程在调用段的外部, 那么它必须表示成 PUBLIC, 而且应当使用 CALL 进入它。

对于 .EXE 程序, 作为执行入口点的主 PROC 必须是 FAR。

(30) 处理器伪操作。这些伪操作定义汇编程序要识别的处理器。处理器伪操作可以放在程序的开始, 或放在你希望处理器特性被允许或被禁止的地方。默认模式是 8086。

- .286, .386, .486, 以及 .586 允许所有低于被命名处理器的指令系统有效(例如, .386 允许 .286 和 .8086。)
- .286P, .386P, .486P, 以及 .586P 允许刚才提到的所有指令系统以及处理器特权指令。

(31) PUBLIC 伪操作。PUBLIC 伪操作通知汇编程序与连接程序: 在汇编中它所标识的符号是被与当前模块连接的另一模块所引用的。格式是 PUBLIC symbol [, ...], 其中 symbol(符号)可以是标号, 数(在 2 字节的范围内)或变量。参见“EXTRN 伪操作”一节和第 22 章的例子。

(32) RECORD 伪操作。RECORD 伪操作允许定义位模式, 比如颜色模式和一位或多位开关指示器。它的格式是

record-name	RECORD	field-name:width[-exp] [, ...]
-------------	--------	--------------------------------

record-name(记录名)和 field-name(字段名)可以是任何唯一的有效标识符。跟在每个字段名后的是冒号(:)和宽度(width)(它指定位数)。宽度项的范围是 1 到 32 位。长度达到 8 变成 8 位, 9 到 16 变成 16 位, 而 17 到 32 则变成 32 位, 如有必要, 还要有内容的右对准。下面的例子使用 RECORD 伪操作定义 BIT_REC:

```
BIT_REC RECORD BITS1: 3, BITS2: 7, BITS3: 6
```

BITS1 定义 BIT_REC 的前面 3 位, BITS2 定义接下去的 7 位, BITS3 定义最后 6 位。总数是 16 位, 或一个字。可以把记录中的值初始化如下:

```
BIT_REC2 RECORD BITS1: 3=101B, BITS2: 7=0110110B, BITS3: 6=011010B
```

注意, RECORD 定义不进行任何存储。因此, 在数据段中, 跟着 RECORD 定义, 必须编写另一个语句为记录分配存储器。定义唯一有效的名字, 记录名, 以及一对尖括号(小于和大于符号)组成的操作数:

```
DEF_BITS BIT_REC <>
```

在数据段中对 DEF_BITS 的分配产生目标码 AD9AH(按 9AAD 存放)。该尖括号还可以包含再定义 BIT_REC 的项。

图 25-1 的部分程序把 BIT_REC 定义为 RECORD, 但记录字段没有初始值。在这种情况下, 数据段中的分配语句就像尖括号内所示的那样去初始化每个字段。

记录专用的操作符是 WIDTH, shift count, 以及 MASK。使用这些操作符允许修改 RECORD 定义而不必修改引用它的指令。

① **WIDTH 操作符**。WIDTH 操作符回送宽度, 该宽度是 RECORD 或 RECORD 字段的位数。例如, 图 25-1 中, 跟在标号 A20 后面的是 2 个 WIDTH 的例子。第一个 MOV 回送整个记录 BIT_REC 的宽度(16 位); 第二个 MOV 回送记录字段 BITS2 的宽度(7 位)。在两种情况下, 汇编程序都为宽度产生了立即操作数。

② **Shift count(移位计数)**。直接引用 RECORD 字段, 比如 MOV CL, BITSB, 是不能引用 BITSB 内容的。替代的办法是: 汇编程序产生一个包含移位计数的立即操作数分隔该字段。该立即数表示必须把 BITSB 移位到右对准的位数。图 25-1 中, 跟在 A30 后面的 3 个例子分别回送 BITSA、BITSB 和 BITSC 的移位计数值。

③ **MASK 操作符**。MASK(屏蔽)操作符回送屏蔽, 含有 1 的位表示指定字段, 实际上它定义了该字段占有的位的位置。例如, BIT_REC 中所定义的每个字段的屏蔽是:

字段	二进制	十六进制
BITSA	1110900000000000	E000
BITSB	0001111111000000	1FC0
BITSC	0000000000111111	003F

图 25-1 中, 跟在 A40 后面的 3 条指令回送 BITSA、BITSB 和 BITSC 的屏蔽值。跟在 A50 和 A60 后面的指令分别把 BISTB 和 BISTA 与 BIT_REC 分隔开来。A50 把该记录取到 AX 中并使用 BITSB 的屏蔽与之相与:

```
记录:      101 0110110 011010
AND BITSB 的 MASK: 000 1111111 000000
结果:      000 0110110 000000
```

结果是除了 BISTB 那些位外, 清除所有位。接下来的 2 条指令使 AX 移 6 位, 这样 BITSB 被右对准为:

```
00000000000110110 (0036H)
```

跟在 A60 后面的例子把记录取到 AX 中, 由于 BITSA 是最左边的字段, 例行程序只是使用移位因子右移 13 位:

```
0000000000000101 (0005H)
```

(33) **SEGMENT 伪操作**。汇编模块由一个或多个段、段的一部分或几个段的相同部分组成。段的格式是

segment-name	SEGMENT	[align][combine]['class']
	...	
segment-name	ENDS	

align, combine 和 class 操作数是可选的。

```

        .DATA
        BIT_REC RECORD BITSA:3,BITSB:7,BITSC:6 ; 定义
        DEF_BITS BIT_REC <101B,0110110B,011010B> ; 初始化
        .CODE
        ...
A20:    ; 宽度:
        MOV     BH,WIDTH BIT_REC ; 记录的宽度 (16)
        MOV     AL,WIDTH BITSB ; 字段的宽度 (07)
A30:    ; 移位计数值:
        MOV     CL,BITSA ; hex 0D
        MOV     CL,BITSB ; 06
        MOV     CL,BITSC ; 00
A40:    ; 屏数:
        MOV     AX,MASK BITSA ; hex E000
        MOV     EX,MASK BITSB ; 1FC0
        MOV     CX,MASK BITSC ; 003F
A50:    ; 分离 BITSB:
        MOV     AX,DEF_BITS ; 取记录
        AND     AX,MASK BITSB ; 清除 BITSB 和 C
        MOV     CL,BITSB ; 取移位 06
        SHR     AX,CL ; 右移
A60:    ; 分离 BITSA:
        MOV     AX,DEF_BITS ; 取记录
        MOV     CL,BITSA ; 取移位 13
        SHR     AX,CL ; 右移
        ...

```

图 25-1 使用 RECORD 伪操作

① **Align(对准)**。这一操作数指明段的起始边界:

BYTE 下一个地址
WORD 下一个偶地址(可被 2 整除的)
DWORD 下一个双字地址(可被 4 整除的)
PARA 下一个小段(可被 16 或 10H 整除的)
PAGE 下一个页地址(可被 256 或 100H 整除的)

PARA 通常用于所有类型的段。BYTE 和 WORD 可以用于与其他段组合的段,通常是数据段。

② **Combine(组合)**。组合操作数 NONE, PUBLIC, STACK, 以及 COMMON 指定连接程序处理段的方法:

- **NONE(默认的)**: 该段在逻辑上是与其他段分隔开的,然而它可以物理上与其他段相邻。该段假定有自己的基地址。
- **PUBLIC: LINK** 把同名和同类别的 PUBLIC 段相互邻接地装入。对于所有这样的 PUBLIC 段假定只有一个基地址。
- **STACK: LINK** 对 STACK 的处理和 PUBLIC 一样。在被连接的 EXE 程序中,至少必须定义一个堆栈。如果有一个以上的堆栈,则 SP 和第一个堆栈相关联。
- **COMMON**: 如果 COMMON 段具有相同的名字和类别,那么连接程序给它们同样的基地址。在执行期间,第二个段覆盖到第一个段上。即使最大的段被覆盖,它还是决定了公用区的长度。
- **AT 小段地址**: 小段必须预先定义。该项便于在存储器的固定区域内的固定偏移地址处定义标号和变量,比如在存储器低端的中断表或 40[0]H 处的 BIOS 数据区。例如,定义视频显示区的单元为:

```
VIDEO_RAM SEGMENT AT 0B800H
```

汇编程序建立一个虚段，它实际上提供了存储单元的映像。

③ ‘Class (类别)」。这一项目帮助连接程序把不同名的段联系起来，识别段，以及控制它们的次序。类别可以包含任何单引号中的有效名字。连接程序使用此名字把有相同名字和类别的段联系在一起。典型的例子是 ‘Data’ 和 ‘Code’。如果定义一个类别 ‘Code’，那么连接程序希望那个段里包含指令码。另外，CODEVIEW 调试程序希望类别 ‘Code’ 作为代码段。

连接程序把下面 2 个具有同名(CSEG)和同类别(‘Code’)的段组合在一起成为在同一段寄存器中的物理段：

```

module1      CSEG      SEGMENT  PARA  PUBLIC  'Code'
              ASSUME    CS: CSEG
              ...
              CSEG      ENDS
              -----
module2      CSEG      SEGMENT  PARA  PUBLIC  'Code'
              ASSUME    CS: CSEG
              ...
              CSEG      ENDS

```

为了控制一个程序内段的排列次序，了解连接程序如何处理这一过程是有用的。段名的原始次序提供了基本的序列，借助于 PUBLIC 属性和类别名可以不考虑这个序列。下面的例子说明了连接前的 2 个目标模块(2 个模块都包含名为 DATASEG1 的段，这两个段具有 PUBLIC 属性和相同的类别名)：

```

module 1      STACK      SEGMENT  PARA      STACK
module 1      DATASEG1   SEGMENT  PARA      PUBLIC  'Data'
module 1      DATASEG2   SEGMENT  PARA
module 1      CODESEG      SEGMENT  PARA      'Code'
module 2      DATASEG1   SEGMENT  PARA      PUBLIC  'Data'
module 2      DATASEG2   SEGMENT  PARA
module 2      CODESEG      SEGMENT  PARA      'Code'

```

在.OBJ 模块被连接以后，.EXE 模块看来就像这样：

```

module 1      CODESEG      SEGMENT  PARA      'Code'
module 2      CODESEG      SEGMENT  PARA      'Code'
modules 1+2    DATASEG1   SEGMENT  PARA      PUBLIC  'Data'
module 1      DATASEG2   SEGMENT  PARA
module 2      DATASEG2   SEGMENT  PARA
module 1      STACK      SEGMENT  PARA      STACK

```

可以嵌套段，一个被嵌套的段完全包含在另一个段之内。下面的例子中，DATASEG2 是完全包含在 DATASEG1 中的：

```

DATASEG1      SEGMENT
...
DATASEG2      SEGMENT
...
DATASEG2      ENDS
...

```

DATASEG1 再继续

```
DATASEG1    ENDS
```

.ALPHA, .SEQ, 以及 DOSSEG 伪操作和汇编程序选项/A 与/S 也可以控制段的次序 (把各段组合成组, 见 GROUP 伪操作)。

(34) **.SEQ 伪操作**。这一伪操作(默认的)放在程序起点上或附近, 它告诉汇编程序保留段的原始序列。也可以使用汇编程序命令行选项/A(参见.ALPHA 和 DOSSEG 伪操作)。

(35) **.STACK 伪操作**。这个简化段伪操作定义堆栈。它的格式是 STACK [size], 其中默认的堆栈大小是 1024 个字节, 它是可以自行定义的(参见.MODEL 伪操作)。

(36) **.STRATUP 伪操作**。可以在代码段的起点使用这个伪操作初始化 DS、SS 和 SP。参见.EXIT 伪操作。

(37) **STRUC/STRUCT 伪操作**。STRUC 伪操作(MASM 6.0 以后是 STURCT)便于定义结构内的相关字段。它的格式是

```
structure-nameSTRUC/STRUCT
...
[所定义的字段]
...
structure-name ENDS
```

一个结构用它的名字和伪操作 STRUC 开始, 并且用它的名字和伪操作 ENDS 结束。汇编程序识别从该结构起点开始(一个接一个地)所定义的字段。有效的项是 DB, DW, DD, DQ, 以及 DT, 它们是用可选的字段名定义的。

在图 25-2 的部分程序中, STRUC 定义一个名为 PARAMLIST 的参数表, 它是用于使用 INT 21H 的功能 0AH 去通过键盘输入一个名字。注意, 像 RECORD 伪操作一样, STRUC 实际上不进行任何存储操作。为了给结构分配存储区, 分配语句是必需的, 使得结构在程序中是可寻址的:

```
PARAMS    PARAMLIST < >
```

这个例子中的尖括号是空的, 但是可以利用它们去再定义(或废弃)在结构内的数据。

指令可以用结构名直接引用该结构。为了引用在结构中的字段, 指令必须根据所用的结构分配名(在例子中是 PARAMS), 接着用一个点把它与字段名连接起来形成一个整体。例如, MOV AL, PARAMS.ACTLEN。

还可以使用分配语句(在图 25-2 中的 PARAMS)去重定义在结构内的字段内容。

(38) **SUBTTL/SUBTITLE 伪操作**。SUBTTL 伪操作(MASM 6.0 以后是 SUBTITLE)把最多达 60 个字符的子标题打印在汇编源列表每页的行 3 上。可以任意次地编写这一伪操作。它的格式是 SUBTTL/SUBTITLE text。

(39) **TEXTEQU 伪操作**。这一伪操作(由 MASM 6.0 引入的)的格式是 TEXTEQU [text-item]。操作数 text-item(文本项)可以用字母表示的串, %开头的常数, 或宏功能回送的串。

(40) **TITLE 伪操作**。TITLE 伪操作把最多达 60 个字符的标题打印在源列表每页的行 2 上。只能按照格式 TITLE text 在起点上编写 TITLE 一次。

(41) **.XCREF/.NOCREF 伪操作**。.XCREF 伪操作(MASM 6.0 以后是.NOCREF)通知汇编程序压缩交叉引用表。它的格式是.XCREF [name[, name]...]。省略操作数使表中的所有项

目都被压缩。也可以只压缩交叉引用的特殊项目。下面是.XCREF 和.CREF 的例子：

```
.XCREF                ; 压缩所有交叉引用
...
.CREF                 ; 恢复所有交叉引用
...
.XREF FIELDA, FIELDB ; 压缩两项交叉引用
```

```

        .DATA
PARAMLIST STRUC                ; 参数表
MAXLEN   DB    25              ;
ACTLEN   DB    ?               ;
NAMEIN   DB    25 DUP(' ')    ;
PARAMLIST ENDS

PARAMS   PARAMLIST <>          ; 分配存储区
PROMPT   DB    'What is the part no.?'
; -----
        .CODE
        ...
        MOV     AH, 40H         ; 请求显示
        MOV     BX, 01
        MOV     CX, 21          ; 提示符长度
        LEA     DX, PROMPT      ; 提示符地址
        INT     21H
        MOV     AH, 0AH         ; 接受键盘
        LEA     DX, PARAMS      ; 输入
        INT     21H
        MOV     AL, PARAMS.ACTLEN
;
        ...                    ; 输入长度

```

图 25-2 使用结构

(42) **.XLIST/.NOLIST** 伪操作。可以使用.XLIST 伪操作(MASM 6.0 以后是.NOLIST)在源程序的任何地方中止对被汇编程序的列表。典型的位置是与其他程序公用的那些语句所在地，以及不需要其他列表的地方。.LIST 伪操作(默认的)恢复列表。使用这些伪操作不带操作数。

目的：解释机器码，介绍 PC 指令系统

26.1 引言

这一章解释机器码并提供一个带有用途说明的符号指令列表。

许多指令都有它特定的用途，以至于 1 字节机器语言指令代码已经足够用了。下面是一些例子：

机器码	符号指令	注释
40	INC AX	； AX 增量
50	PUSH AX	； AX 进栈
C3	RET(short)	； 从过程短返回
CB	RET(far)	； 从过程远返回
FD	STD	； 设置方向标志

这些指令都无法直接访问存储器。指定一个立即操作数，两个寄存器，或访问存储器的指令要复杂一些，并且要求 2 个或更多的机器码字节。

机器码对于指明特定寄存器有专门的规定，并对用寻址方式字节访问存储器有另外的规定。

26.2 寄存器表示法

访问寄存器的指令可以有指定寄存器的 3 位并可以有指明宽度是字节(0)还是字(1)的 w 位。而且，只有某些指令可以访问段寄存器。图 26-1 说明寄存器的表示法。例如，当 w 位为 0 时，位值 000 指的是 AH；而 w 位为 1 时，则指的是 AX。

这里是具有 1 字节立即操作数的 MOV 指令的符号与机器码：

```
MOV AH, 00      10110 100 00000000
                  |  | | |
                  w reg=AH
```

在这种情况下，机器码的第一个字节指明一个字节的宽度($w=0$)，并指的是 AH(100)。下

面是包含一个字立即操作数的 MOV 指令，以及它所产生的机器码：

```
MOV AX, 00      10111 000 00000000 00000000
                  | 1111
                  w reg=AX
```

机器码的第一个字节指明一个字的宽度(w=1)，并指的是 AX(000)。对于其他指令，w 和 reg 可能占有不同位置。机器码的第一个字节指明流向(左/右)的 d 位。

机器码练习。试使用调试程序进行以下有关机器码的练习(对于 DEBUG，使用命令 A 100)。按这样的次序输入 MOV reg, 0 指令：AL, CL, DL, BL, AH, CH, DH, BH, AX, CX, DX, BX, SP, BP, SI，以及 DI。注意机器码是如何从 B0 到 BF 增量的。

26.3 寻址方式字节

当寻址方式字节存在时，会占有机器码的第二个字节，并由以下 3 个元素组成：
 mod 2 位，表示方式，其中值 00、01 和 10 指的是存储单元，而 11 指的是寄存器。
 reg 3 位，引用寄存器。
 r/m 3 位，引用寄存器或存储器，其中 r 指的是寄存器，而 m 指明存储器地址。

通用，基址和变址寄存器			段寄存器	
位	w = 0	w = 1	位	
000	AL	AX/EAX	000	ES
001	CL	CX/ECX	001	CS
010	DL	DX/EDX	010	SS
011	BL	BX/EBX	011	DS
100	AH	SP	100	FS
101	CH	BP	101	GS
110	DH	SI		
111	BH	DI		

图 26-1 寄存器表示法

在以下把 AX 加到 BX 的例子中：

```
ADD BX, AX      00000011 11 011 000
                  11 11 111 111
                  dw mod reg r/m
```

d=1 的意思是 mod(11)和 reg(011)描述第一个操作数，而 r/m(000)描述第二个操作数。由于 w=1，所以宽度是一个字。因此，该指令是把 AX(000)加到 BX(011)中。

目标码的第二个字节说明了大多数存储器寻址方式。可以使用 DEBUG 检验该例子：键入机器码为 E 100 03 D8 并用 U 100 101 来反汇编它。

Mod 位。2 个 mod 位区分是寄存器寻址还是存储器寻址。下面说明它们的用途：

- 00 r/m 位给出严格的寻址选项；无偏移字节(除 r/m=110 之外)。
- 01 r/m 位给出严格的寻址选项；一个偏移字节。
- 10 r/m 位给出严格的寻址选项；两个偏移字节。

11 r/m 指定寄存器。w 位(在操作码字节中)决定访问的是 8 位、16 位或 32 位的寄存器。

Reg 位。3 个 reg 位与 w 位相结合来确定实际宽度。

R/M 位。3 个 r/m(寄存器/存储器)位与 mod 位相结合来确定寻址方式，如图 26-2 所示：

r/m	mod=00	mod=01 或 10	mod=11 w=0	mod=11 w=1
000	BX+SI	DS:[BX+SI+disp]	AL	AX
001	BX+DI	DS:[BX+DI+disp]	CL	CX
010	BP+SI	SS:[BP+SI+disp]	DL	DX
011	BP+DI	SS:[BP+DI+disp]	BL	BX
100	SI	DS:[SI+disp]	AH	SP
101	DI	DS:[DI+disp]	CH	BP
110	Direct	SS:[BP+disp]	DH	SI
111	BX	DS:[BX+disp]	BH	DI

图 26-2 r/m 位

26.3.1 两字节指令

以下两字节指令把 BX 加到 AX 上：

```
ADD AX, BX      0030 0011 11 000 011
                  11 11 111 111
                  dw mod reg r/m
```

- d=1 reg 加上 w 描述第一个操作数(AX)，而 mod 加上 r/m 再加上 w 描述第二个操作数(BX)。
- w=1 宽度是一个字。
- mod=11 第二个操作数是寄存器。
- reg=000 第一个操作数是 AX。
- r/m=011 第二个操作数是 BX。

下一个例子是 AL 乘以 BL：

```
MUL BL          11110110 11 100 011
                  1 11 111 11
                  w mod reg r/m
```

宽度(w=0)是一个字节，mod(11)引用寄存器，寄存器(r/m=011)是 BL。Reg=100 在这里没有意义。如果乘数是一个字节(如在本例中)，那么处理器认为被乘数在 AL 中；如果乘数是一个字，则认为被乘数在 AX 中；如果乘数是双字，则认为被乘数在 EAX 中。

26.3.2 三字节指令

以下 MOV 产生三字节的机器码：

```
MOV mem-word, AX    10100011 00000000 00000000
                      11
                      dw
```

来自累加器(AX 或 AL)的传送仅仅需要知道操作是字节还是字。在这个例子中, $w=1$ 的意思是字, 所以采用 16 位 AX(第二个操作数中如编码为 AL, 则将使 w 位成为零)。字节 2 和字节 3 包含对于存储单元的偏移值。使用累加器会比其他寄存器产生较短的指令长度和较快的执行速度。

26.3.3 四字节指令

以下四字节指令是 AL 乘以一个存储单元:

```
MUL mem-byte    11110110 00 100 110 00000000 00000000
                  1 11 111 111
                  w mod reg r/m
```

对于这条指令, 虽然 reg 是 100, 但还是假定被乘数是 AL(一个字节, 因为 $w=0$)。Mod=00 指明是存储器访问, 而 $r/m=110$ 的意思是直接引用存储器, 2 个后继的字节提供对于存储单元的偏移值。

下面的例子说明 LEA 指令, 它指定一个字地址:

```
LEA DX, memory    10001101 00 010 110 00000000 00000000
                    1 1 111 111
                    LEA mod reg r/m
```

$reg=010$ 指明 DX; mod=00 和 $r/m=110$ 指明直接引用存储器地址; 后面 2 个字节提供对于这一单元的偏移值。

26.4 指令系统

这一节涉及按字母顺序排列的指令系统。为了方便起见, 一些密切相关的指令(如条件转移)是组合在一起的。80286 和后继的处理器支持的许多特殊指令没有在这里介绍, 它们是: ARPL, BOUND, CLTS, LAR, LGDT, LIDT, LLDT, LMSW, LSL, LTR, SGDT, SIDT, SLDT, SMSW, STR, VERR, 以及 VERW。80486 及其后继型号特有的指令: BSWAP, INVD, WBINVD, 以及 INVLPG 也没有包括在内。对双字寄存器或存储单元的引用包含在 80386 或后继的处理器中。

除前面讨论的方式字节和宽度位外, 以下缩写词是有关系的:

addr	存储单元的地址。
addr-high	地址的最右边字节。
addr-low	地址的最左边字节。
data	立即操作数(如 $w=0$, 是 8 位; 如 $w=1$, 是 16 位)。
data-high	立即操作数的最右边字节。

data-low 立即操作数的最左边字节。
 disp 位移量(偏移值)。
 reg 引用的寄存器。

标志的缩写词如下: AF=辅助, CF=进位, DF=方向, IF=中断, OF=溢出, PF=奇偶, SF=符号, TF=陷阱, 以及 ZF=零。

26.4.1 AAA: 加法后的 ASCII 调整

校正 2 个 ASCII 字节在 AL 中的和(ADD 之后的)。如果 AL 最右边 4 位的值大于 9, 或者如果 AF 被置成 1, 则 AAA 将 AH 加 1, AL 加 6, 并使 AF 与 CF 置 1。否则, 清除 AF 与 CF。

AAA 总是清除 AL 最左边的 4 位。

标志: 影响 AF 和 CF(OF, PF, SF 和 ZF 未定义)。

源码: AAA(无操作数)。

目标码: 00110111

26.4.2 AAD: 除法前的 ASCII 调整

在除法之前, 调整在 AX 中的非压缩 BCD 值(被除数)。AAD 将 AH 乘以 10, 把乘积加到 AL 上并清除 AH。在 AX 中结果的二进制值, 现在等效于原来的非压缩 BCD 值, 并为二进制除法操作准备就绪。

标志: 影响 PF, SF, 以及 ZF(AF, CF 和 OF 未定义)。

源码: AAD(无操作数)

目标码: 111010101000010101

26.4.3 AAM: 乘法后的 ASCII 调整

调整由 2 个非压缩 BCD 数字相乘所产生的在 AL 中的乘积。AAM 将 AL 除以 10, 并把商放在 AH, 把余数放在 AL 中。

标志: 影响 PF, SF 和 ZF(AF, CF 和 OF 未定义)。

源码: AAM(无操作数)

目标码: 1110101001000010101

26.4.4 AAS: 减法后的 ASCII 调整

调整在 AL 中的 2 个 ASCII 字节差(SUB 之后)。如果最右边 4 位的值大于 9, 或者进位标志是 1, 则 AAS 从 AL 中减 6, 从 AH 中减 1, 并使 AF 和 CF 置 1。否则, 清除 AF 和 CF。AAS 总是清除 AL 的最左边 4 位。

标志: 影响 AF 和 CF(OF, PF, SF 和 ZF 未定义)。

源码: AAS(无操作数)

目标码: 00111111

26.4.5 ADC: 带进位加法

常用于二进制多字加法, 把溢出的 1 的位带到算术运算的下一级。ADC 把进位标志的内容(0/1)加到操作数 1, 然后就像 ADD 一样, 把操作数 2 加到操作数 1 上(参见 SBB)。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: ADC register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 reg:	1000100dw modreg r/m
immed(立即数)到 accumulator(累加器):	10000010w --data-- data 如 w=11
immed 到 reg/mem:	1100000sw mod010r/m --data-- data 如 sw=011

26.4.6 ADD: 二进制数加法

把来自存储器、寄存器或立即数的二进制值加到寄存器, 或者把寄存器或立即数的值加到存储器。值可以是字节, 字或双字。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: ADD register/memory, register/memory/immediate

目标码: 3 种格式:

regmem 与 reg:	1000000dw modreg r/m
immed 到 accumulator:	10000010w --data-- data 如 w=11
immed 到 reg/mem:	1100000sw mod000r/m --data-- data 如 sw=011

26.4.7 AND: 逻辑与

完成 2 个操作数按位逻辑与操作。2 个操作数必须同时是字节, 字或双字, AND 要求位与位的匹配。对于每一对相匹配的位都是 1 时, 在第一个操作数中的 1 的位被置成 1, 否则该位被清除。(参见 OR, XOR, 以及 TEST。)

标志: 影响 CF(0), OF(0), PF, SF 和 ZF(AF 未定义)。

源码: AND register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 reg:	1001000dw modreg r/m
immed 到 accumulator:	10010010w --data-- data 如 w=11
immed 到 reg/mem:	1100000sw mod100r/m --data-- data 如 w=11

26.4.8 BSF/BSR: 位正向扫描/位反向扫描(80386+)

为了第一个是 1 的位, 对操作数 2 的位串(16 或 32 位)进行扫描。BSF 是从右向左扫描, 而 BSR 则是从左向右扫描。如果找到了是 1 的位, 则操作把它的位置(相对于右边的位 0)回送到操作数 1 的寄存器中, 并把零标志置 0; 否则, 该位被置 1。

标志: 影响 ZF。

源码: BSF/BSR register, register/memory

目标码: BSF: 1000011111101111001modreg/r/m1

BSR: 1000011111101111011modreg/r/m1

26.4.9 BT/BTC/BTR/BTS: 位测试(80386+)

把指定的位复制到进位标志。操作数 1 包含被测试的位串, 操作数 2 包含指明该位位置的值。BT 简单地把该位复制到 CF。其他指令也复制该位, 同时还会影响操作数 1 中对应的位: BTC 用将操作数 1 中该位的值变反的方法对位求反; BTR 用把该位清零的办法进行复位; BTS 把该位置成 1。引用是对 16 或 32 位值进行的。

标志: 影响 CF。

源码: BT/BTC/BTR/BTS register/memory, register/immedicete

目标码: 2 种格式:

immed 对 reg: 1000011111101110101modr***r/m1

reg/mem 对 reg: 100001111110***0101modreg/r/m1

(***的意思是 100=BT, 111=BTC, 110=BTR, 101=BTS)

26.4.10 CALL: 调用一个过程

调用一个近的或远的过程。如果被调用的过程是 NEAR, 汇编程序产生一个近 CALL; 如果被调用过程是 FAR, 则产生一个远 CALL。近 CALL 使 IP 进栈(下一条指令的偏移值), 然后把目的偏移值装入 IP。远 CALL 使 CS 进栈并把目的段地址装入 CS, 然后使 IP 进栈并把目的偏移值装入 IP。在返回时, 后继的 RETN 或 RETF 则用来把这些步骤反过来进行。

标志: 不受影响。

源码: CALL register/memory/precedure-name

目标码: 4 种格式:

段内直接: 1111010001disp-low1disp-high1

段内间接: 1111111111mod010r/m1

段间间接: 1111111111mod011r/m1

段间直接: 1100110101offset-low1offset-high1seg-low1seg-high1

26.4.11 CBW: 字节转换为字

用把 AL 的符号(位 7)复制到 AH 所有位的方法, 把 1 字节带符号值扩展为带符号的字。
(还可见于 MOVSX)

标志: 不受影响。

源码: CBW(无操作数)

目标码: 10011000

26.4.12 CDQ: 双字转换为四字(80386+)

用把 EAX 的符号(位 31)复制到整个 EDX 的方法, 把 32 位带符号值扩展为 64 位带符号的值。

标志: 不受影响。

源码: CDQ(无操作数)

目标码: 10011001

26.4.13 CLC: 清除进位标志

清除 CF, 例如, 可连 ADC 不加 1(参见 STC)。

标志: CF(置 0)。

源码: CLC(无操作数)

目标码: 11111000

26.4.14 CLD: 清除方向标志

清除 DF, 使得如 MOVS 那样的串操作从左到右进行处理(参见 STD)。

标志: DF(置 0)

源码: CLD(无操作数)

目标码: 11111100

26.4.15 CLI: 清除中断标志

清除 IF, 禁止可屏蔽的外部中断(参见 STI)。

标志: IF(置 0)。

源码: CLI(无操作数)

目标码: 11111010

26.4.16 CMC: 进位标志变反

CF 位的值变反: 0 变成 1, 1 变成 0。

标志: CF(反向)。

源码: CMC(无操作数)

目标码: 11110101

26.4.17 CMP: 比较

比较 2 个数据字段的二进制内容。CMP 从操作数 1 中减去操作数 2, 并设置/清除标志, 但不存结果。而 2 个操作数都是字节、字或双字。CMP 可以进行寄存器、存储器或立即数与寄存器的比较, 或者进行寄存器或立即数与存储器的比较(CMP 做数值比较, 串比较见 CMPS)。结果如下:

	CF	SF	ZF
操作数 1 < 操作数 2	1	1	0
操作数 1 = 操作数 2	0	0	1
操作数 1 > 操作数 2	0	0	0

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: CMP register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 reg: 1001110dw!modreg/r
immed 对 accumulator: 10011110w!--data--!data 如 w=11
immed 对 reg/mem: 1100000sw!mod11!r/m!--data--!data 如 sw=0

26.4.18 CMPS/CMPSB/CMPSW/CMPSD: 串比较

对存储器中任意长度的串进行比较。REPn 前缀通常和 CX 中的最大值一起放在这些指令的前面。CMPSB 比较字节, CMPSW 比较字, 而 CMPSD(80386+)则是比较双字。DS: SI 寻址操作数 1, ES: DI 寻址操作数 2。如果方向标志是 0, 则比较操作从左到右进行并且 SI 与 DI 加 1(对于字节), 加 2(对于字), 以及加 4(对于双字); 如果 DF 是 1, 则比较从右到左进行并使 SI 与 DI 减量。REPn 对于每次重复都使 CX 减 1。REPNE 第一次发现匹配时, 结束操作; REPE 第一次发现不匹配时, 结束操作; 或者当 CX 减到 0 时, 二者均结束操作。在结束时, DI 与 SI 前进了一步, 超过了产生结束的字节。最后的比较要设置/清除标志。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: [REPnn] CMPSB/CMPSW/CMPSD(无操作数)

目标码: 1010011w

26.4.19 CMPXCHG: 比较并交换(80486+)

比较累加器(AL, AX 或 EAX)和操作数 1。如果相等, 则 CMPXCHG 把操作数 2 复制到操作数 1, 并使零标志置 1; 如果不相等, 则 CMPXCHG 把操作数 1 复制到累加器 2 并清除 ZF。操作数 1 和操作数 2 被编码, 而累加器的第 3 个元素则不编码。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: CMPXCHG register/memory, register

目标码: Hex 0F B0/r 或 0F B1/r

26.4.20 CMPXCHG8B: 比较并交换(Pentium+)

比较 64 位的 EDX: EAX 和操作数。如果相等, CMPXCHG8B 把 EDX: EAX 装入到操作数中, 并使零标志置 1; 如果不相等, 则把操作数装入到 EDX: EAX, 并清除 ZF。

标志: 影响 ZF。

源码: CMPXCHG8B register/memory (一个操作数, 64 位)。

目标码: Hex 0F C7

26.4.21 CWD: 字转换为双字

用把 AX 的符号(位 15)复制到整个 DX 的方法, 把一个字的带符号值扩展为在 DX: AX 中的带符号双字, 通常用来产生一个 32 位的被除数(参见 CBWDE 和 MOVSX)。

标志: 不受影响。

源码: CWD(无操作数)

目标码: 10011001

26.4.22 CWDE: 字转换为扩展的双字(80386+)

用复制 AX 的符号(位 15)的方法, 把一个字的带符号值扩展为 EAX 中的双字, 典型地用来产生一个 32 位的被除数(参见 CWD 和 MOVSX)。

标志: 不受影响。

源码: CWDE(无操作数)

目标码: 10011000

26.4.23 DAA: 加法后的十进制调整

在 ADD 或 ADC 把 2 个压缩的 BCD 项相加之后, 校正在 AL 中的结果。如果最右边 4 位的值大于 9, 或辅助标志(AF)是 1, 则 DAA 使 AL 加 6 并使 AF 置 1。如果在 AL 中的值大

于 99H，或如果进位标志(CF)是 1，则 DAA 把 60H 加到 AL 中并使 CF 置 1。否则，清除 AF 与 CF。现在，AL 包含一个正确的 2 个数位的压缩十进制结果(参见 DAS)。

标志：影响 AF，CF，PF，SF 和 ZF(OF 未定义)。

源码：DAA(无操作数)

目标码：00100111

26.4.24 DAS：减法后的十进制调整

在 SUB 或 SBB 把 2 个压缩的 BCD 项相减之后，校正在 AL 中的结果。如果最右边 4 位的值大于 9，或辅助标志(AF)是 1，则 DAS 使 AL 减去 6，并使 AF 置 1。如果 AL 中的值大于 99H 或进位标志(CF)是 1，则 DAS 从 AL 中减去 60H，并使进位标志置 1。否则，清除 AF 和 CF。现在 AL 包含一个正确的 2 个数位的压缩十进制值(参见 DAA)。

标志：影响 AF，CF，PF，SF 和 ZF (OF 未定义)。

源码：DAS(无操作数)

目标码：00101111

26.4.25 DEC：减 1

从寄存器或存储器的一个字节、字或双字中减 1，并把该值看成是一个无符号整数(参见 INC)。

标志：影响 AF，OF，PF，SF 和 ZF。

源码：DEC register/memory

目标码：2 种格式：

register: 101001regl
reg/memory: 1111111w mod001r/ml

26.4.26 DIV：无符号数除法

在操作数 1 中的无符号被除数除以无符号除数。DIV 把最左边的 1 位看作是数据位，没有负的符号。除以零会引起除以零中断(参见 IDIV)。以下是按被除数大小而确定的除法操作：

大小	被除数 (操作数 1)	除数 (操作数 2)	商	余数	举例
16 位	AX	8 位 寄存器/存储器	AL	AH	DIV BH
32 位	DX: AX	16 位 寄存器/存储器	AX	DX	DIV CX
64 位	EDX: EAX	32 位 寄存器/存储器	EAX	EDX	DIV ECX

标志：影响 AF，CF，OF，PF，SF 和 ZF(全部未定义)。

源码：DIV register/memory

目标码：1111011w1mod110r/ml

26.4.27 ENTER: 建立堆栈帧(80286+)

建立堆栈帧, 用于被调用的过程接收传送的参数。操作数 1 给出以字节为单位的堆栈帧大小, 操作数 2 指明嵌套层(对于 BASIC, C, 以及 FORTRAN 编译程序而言, 嵌套层为 0)。该操作使 BP 进栈, 把 SP 传送到 BP, 并且从 SP 中减去帧大小。(见作为互补指令的 LEAVE)。

标志: 不受影响。

源码: ENTER size, nesting-level

目标码: `1110010001--data--1--data-`

26.4.28 HLT: 进入暂停状态

在等待硬件中断时, 使处理器进入暂停状态。当一个中断发生时, 处理器使 CS 与 IP 进栈并执行中断例程序。在返回时, IRET 指令使堆栈中的内容出栈, 并且恢复处理紧跟在原来的 HLT 之后的指令(首先必须用 STI 操作使 IF 置 1, 以便允许硬件中断)。

标志: 不受影响。

源码: HLT(无操作数)

目标码: `11110100`

26.4.29 IDIV: 带符号(整数)除法

带符号的被除数除以带符号的除数。IDIV 把最左边的位看作是符号(0=正, 1=负)。除以零会引起除以零中断(见 CBW 和 MOVSX 扩展带符号被除数的长度, 还可参见 DIV)。下面是按照被除数大小而确定的除法操作:

大小	被除数 (操作数 1)	除数 (操作数 2)	商	余数	举例
16 位	AX	8 位 寄存器/存储器	AL	AH	DIV BH
32 位	DX: AX	16 位 寄存器/存储器	AX	DX	DIV CX
64 位	EDX: EAX	32 位 寄存器/存储器	EAX	EDX	DIV ECX

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: IDIV register/memory

目标码: `11111011w1mod111r/m1`

26.4.30 IMUL: 带符号(整数)乘法

带符号的被乘数乘以带符号的乘数。该操作把最左边的位看作符号(0=正, 1=负)。IMUL 支持 4 种格式:

1. 该操作假定被乘数在 AL、AX 或 EAX 中, 并且根据乘数的大小来确定其大小(参见 MUL)。下面是按大小而定的该操作:

大小	被乘数	乘数	乘积	举例
8 位	AL	8 位 寄存器/存储器	AX	IMUL BL
16 位	AX	16 位 寄存器/存储器	DX: AX	IMUL mem-word
32 位	EAX	32 位 寄存器/存储器	EDX: EAX	IMUL ECX

其他 3 种格式引用任何 16 或 32 位的通用寄存器，数据项长度必须相同。

2. 操作数 1(寄存器)包含被乘数，而且所求出的乘积也存放在那里；操作数 2 是个立即值。如：

```
IMUL CX, 32 和 IMUL EBX, 50
```

3. 操作数 1(寄存器)存放求得的乘积；操作数 2(寄存器或存储单元)包含被乘数；操作数 3 是个立即值。如：

```
IMUL CX, DX, 25 和 IMUL EBX, mem-dblword, 100
```

4. 操作数 1(寄存器)包含被乘数，并且存放求得的乘积；操作数 2(寄存器或存储单元)包含乘数。例如：

```
IMUL DX, mem-word 和 IMUL EBX, EDX
```

标志：影响 CF 和 OF(AF, PF, SF 和 ZF 未定义)。

源码：4 种格式

- ① IMUL register/memory (所有处理器)
- ② IMUL register, immediate(80286+)
- ③ IMUL register, register, immediate(80286+)
- ④ IMUL register, register/memory (80386+)

目标码：1111011w1mod101r/m1 (第一种格式)

26.4.31 IN：输入字节或字

从输入端口传送一个字节到 AL，传送一个字到 AX，或传送一个双字到 EAX。端口被编码为固定的数值操作数(如 IN AX, port#)或编码为 DX 中的变量(如 IN AX, DX)。如果端口号大于 256，则使用 DX(参见 INS 和 OUT)。

标志：不受影响。

源码：IN AL/AX, 端口号/DX

目标码：2 种格式：

变量端口：1110110w1

固定端口：11110010w1--port--1

26.4.32 INC：加 1

在寄存器或存储器中的一个字节、一个字或一个双字加 1，并把该值看成是无符号整数，例如编码为 INC ECX(参见 DEC)。

标志：影响 AF, OF, PF, SF 和 ZF(不影响 CF)。

源码: INC register/memory

目标码: 2 种格式:

```
register:      101000regl
reg/memory:   11111111w|mod000r/m|
```

26.4.33 INS/INSB/INSW/INSD: 输入串(80286+)

从端口接收一个串(目的), 其目的是用 ES: DI 寻址, DX 包含端口号。INS_n 通常是和 REP 前缀, 以及包含被接收项目(如字节、字或双字)数的 CX 一起使用的。根据 DF(0/1), 该操作按项的大小使 DI 增量或减量(参见 IN 和 OUTS)。

标志: 不受影响。

源码: [REP] INSB/INSW/INSD(无操作数)

目标码: 10110110w|

26.4.34 INT: 中断

进行中断处理和传送控制给中断向量表中 256 个地址之一。INT 执行以下操作: (1)标志进栈, 并复位中断标志与陷阱标志; (2)CS 进栈, 并把中断地址的高位字存放在 CS 中; (3)IP 进栈, 并用中断地址的低位字填充 IP。对于 80386+来说, INT 对于 16 位段将 16 位的 IP 进栈, 而对于 32 位段则使 32 位 IP 进栈。IRET 用于从中断例行程序返回。

标志: 清除 IF 与 TF。

源码: INT 编号

目标码: 11100110v|--type--| (如 v=0, 则 type 是 3)

26.4.35 INTO: 溢出中断

如果发生溢出(OF 被置 1), 则产生一个中断(通常是无害的), 并且执行 INT 04H。该中断地址是在中断向量表的 10H 单元(参见 INT)。

标志: 影响 IF 与 TF。

源码: INTO(无操作数)

目标码: 11001110

26.4.36 IRET/IRETD: 中断返回

提供从中断例行程序的远返回。IRET 实现以下过程: (1)在栈顶的字出栈进入 IP 中, SP 加 2, 栈顶又出栈进入 CS 中; (2)SP 加 2, 栈顶再出栈进入标志寄存器。这一过程把中断原先采取的步骤恢复原状, 并实现返回。对于 80386+, 使用 IRETD(双字)使 32 位的 IP 出栈(参见 RET)。

标志: 影响所有标志。

源码: IRET(无操作数)

目标码: 11001111

26.4.37 Jcondition: 条件转移

这一节总结了测试各种标志状态的条件转移指令。如果测试结果是真, 该操作把操作数偏移值加到 IP 并传送控制到 CS:IP 地址; 如果不是真, 则按顺序地继续处理下一条指令。对于 8086~80286, 转移必须是短的(-128 到 127 个字节); 对于 80386+, 汇编程序假定为近转移(-32 768 到 32 767 个字节), 但可以使用 SHORT 操作符强制成短转移。该操作测试标志, 但不改变它们。源码是 Jcondition label。所有目标码的格式都是 |distnnnnl--disp--l, 其中 disp 位对于短转移是 0111, 对于近转移是 1000。

在第一张表中, 指令典型地用在比较操作之后, 这是操作数 1 对操作数 2 的比较:

源 码	目 标 码	检查的标志	用于比较之后
JA	dist0111	CF=0, ZF=0	无符号数据, 高于
JAE	dist0011	CF=0	无符号数据, 高于/等于
JB	dist0010	CF=1	无符号数据, 低于
JBE	dist0110	CF=1 或 ZF=1	无符号数据, 低于/等于
JE	dist0100	ZF=1	带符号/无符号数据, 等于
JG	dist1111	ZF=0 或 SF=OF	带符号数据, 大于
JGE	dist1101	SF=OF	带符号数据, 大于/等于
JL	dist1100	SF≠OF	带符号数据, 小于
JLE	dist1110	ZF=1 或 SF≠OF	带符号数据, 小于/等于
JNA	dist0110	CF=1 或 ZF=1	无符号数据, 不高于
JNAE	dist0010	CF=1	无符号数据, 不高于/等于
JNB	dist0011	CF=0	无符号数据, 不低于
JNBE	dist0111	CF=0 或 ZF=0	无符号数据, 不低于/等于
JNE	dist0101	ZF=0	带符号/无符号数据, 不相等
JNG	dist1110	ZF=1 或 SF≠OF	带符号数据, 不大于
JNGE	dist1100	SF≠OF	带符号数据, 不大于/等于
JNI	dist1101	SF=OF	带符号数据, 不小于
JNLE	dist1111	ZF=0 或 SF=OF	带符号数据, 不小于/等于

在第二张表中, 指令通常用在算术运算或其他操作之后, 操作根据结果清除或设置各个位。

源码	目标码	检查的标志	用于测试
JC	dist0010	CF=1	如果 CF 为 1 (同 JB/JNAE)
JNC	dist0011	CF=0	如果 CF 为 0 (同 JAE/JNB)
JNO	dist0001	OF=0	如果 OF 为 0
JNP	dist1011	PF=0	如果没有 (奇) 奇偶性: 在低 8 位中 1 的个数为奇数时设置
JNS	dist1001	SF=0	如果符号是正的
JNZ	dist0101	ZF=0	如果带符号/无符号数据不是零

续表

源码	目标码	检查的标志	用于测试
JC	dist0000	OF=1	如果 OF 为 1
JF	dist1010	PF=1	如果偶奇偶性: 在低 8 位中 1 的个数为偶数时设置
JFE	dist1010	PF=1	同 JP
JFO	disL1011	PF=0	同 JNP
JS	dist1000	SF=1	如果符号是负的
JZ	dist0100	ZF=1	如果带符号/无符号数据是零

26.4.38 JCXZ/JECXZ: 若 CX/ECX 为零则转移

如果 CX 或 ECX 为零, 则转移到指定的地址。该操作在循环的起点可能是有用的, 不过要受短转移的限制。

标志: 不受影响。

源码: JCXZ/JECXZ label

目标码: |11100011|--disp--'

26.4.39 JMP: 无条件转移

在任意条件下, 转移到指定地址。JMP 的地址可能是短的(-128 到+127 个字节), 近的(+32K 或-32K 之内, 默认的), 或远的(转到另一个代码段)。短或近 JMP 用目的偏移地址取代 IP。远转移(如 JMP FAR PTR label)用新的段地址取代 CS; IP。

标志: 不受影响。

源码: JMP register/memory

目标码: 5 种格式:

段内短直接: |11101011|--disp--|

段内直接: |11101001|disp-low'disp-high|

段内间接: |11111111|mod100r/m|

段间间接: |11111111|mod101r/m|

段间直接: |11101010|offset-low|offset-high|seg-low|seg-high|

26.4.40 LAHF: 标志装入 AH

把标志寄存器的低 8 位装入 AH (参见 SAHF)。

标志: 不受影响。

源码: LAHF(无操作数)

目标码: 10011111

26.4.41 LDS/LES/LFS/LGS/LSS: 装入段寄存器

操作数 1 引用任何一个通用、变址或指针寄存器。操作数 2 引用包含偏移: 段地址的存储器中的 2 个字。该操作把段地址装入段寄存器中并把偏移值装入操作数 1 的寄存器中。所用的例子如 `LDS DI, SEG_ADDRESS`。

标志: 不受影响。

源码: `LDS/LES/LFS/LGS/LSS register, memory`

目标码: `LDS: 111000101|modreg|r|m|`

`LES: 111000100|modreg|r|m|`

`LFS: 100001111|10110100|modreg|r|m| (80386+)`

`LGS: 100001111|10110101|modreg|r|m| (80386+)`

`LSS: 100001111|10110110|modreg|r|m| (80386+)`

26.4.42 LEA: 装入有效地址

把近(偏移)地址装入寄存器。

标志: 不受影响。

源码: `LEA register, memory`

目标码: `10001101`

26.4.43 LEAVE: 终止堆栈帧(80286+)

终止由 `ENTER` 操作建立的过程的堆栈帧。`LEAVE` 的动作和 `ENTER` 是相反的(传送 `BP` 到 `SP` 并使 `BP` 出栈)。

标志: 不受影响。

源码: `LEAVE`(无操作数)

目标码: `11001001`

26.4.44 LES/LFS/LGS: 装入附加段寄存器(见 LDS)

26.4.45 LOCK: 封锁总线

防止数值协处理器与处理器同时改变数据项。`LOCK` 是个 1 字节的前缀, 可直接写在任何指令的前面。该操作给协处理器发送一个信号, 禁止协处理器使用数据, 直到下一条指令完成时为止。

标志: 不受影响。

源码: `LOCK 指令`

目标码: `11110000`

26.4.46 LODS/LODSB/LODSW/LODSD: 装入字节串、字串或双字串

把一个值从存储器装入到累加寄存器。虽然 LODS 是个串操作,但它并不要求 REP 前缀。DS:SI 寻址一个字节(如果是 LODSB),寻址字(如果是 LODSW),或寻址双字(如果是 LODSD, 80386+)并分别从存储器装入到 AL、AX 或 EAX 中。如果方向标志是 0,则 SI 加 1(如果是字节)、加 2(如果是字)或加 4(如果是双字);否则,使 SI 减 1、2 或 4。

标志: 不受影响。

源码: LODS mem 或 LODS segreg: mem

LODSB/LODSW/LODSD(无操作数)

目标码: 1010110w

26.4.47 LOOP/LOOPW/LOOPD: 循环直到完成

控制指定次数的例行程序的执行。在开始循环之前 CX 应当包含计数值。LOOP 指令出现在循环的末端并使 CX 减 1。如果 CX 是非零,则 LOOP 传送控制给它的操作数地址(短转移),该地址指向循环的起点(把偏移值加到 IP 中),否则,LOOP 直达下一条指令。

LOOP 在 16 位方式下使用 CX,在 32 位方式下使用 ECX。LOOPW 使用 CX,而 LOOPD(80386+)则使用 ECX。

标志: 不受影响。

源码: LOOPnn label

目标码: 111100010 --disp--1

26.4.48 LOOPE/LOOPZ/LOOPEW/LOOPZW/LOOPED/LOOPZD: 相等/为零时循环

控制例行程序的重复执行。LOOPE 与 LOOPZ 是和 LOOP 类似的,只有当 CX 是非零以及零标志是 1(零条件是由另一条指令设置)时,传送控制给操作数地址(短转移)外,否则,该操作直达下一条指令(参见 LOOPNE/LOOPNZ)。

LOOPE 和 LOOPZ 在 16 位方式下使用 CX,在 32 位方式下使用 ECX。LOOPEW 和 LOOPZW 使用于 CX,而 LOOPED 和 LOOPZD(80386+)使用 ECX。

标志: 不受影响。

源码: LOOPnn label

目标码: 1111000011 --disp--1

26.4.49 LOOPNE/LOOPNZ/LOOPNEW/LOOPNZW: 不相等/不为零时循环

控制例行程序的重复执行。LOOPNE 与 LOOPNZ 是和 LOOP 类似的,除了当 CX 是非

零和零标志是 0(非零条件由另一条指令设置)时传送控制给操作数地址(短转移)外,该操作直达下一条指令(参见 LOOPE/LOOPZ)。

LOOPNE 和 LOOPNZ 在 16 位方式下使用 CX, 在 32 位方式下使用 ECX。LOOPNEW 和 LOOPNZW 使用 CX, 而 LOOPNED 和 LOOPNZD(80386+)使用 ECX。

标志: 不受影响。

源码: LOOPNnn label

目标码: 111100000 --disp--

26.4.50 LSS: 装入堆栈段寄存器(见 LDS)

26.4.51 MOV: 传送数据

在 2 个寄存器或寄存器与存储器之间传送数据, 并可传送立即数据到寄存器或存储器。所引用的数据定义被传送的字节数(1, 2 或 4), 操作数大小必须一致。MOV 不能在 2 个存储单元之间进行传送(使用 MOVS), 也不能把立即数据传送到段寄存器或从段寄存器到段寄存器进行传送(参见 MOVSW/MOVSX)。

标志: 不受影响。

源码: MOV regisect/memory, register/memory/immediate

目标码: 7 种格式:

```
reg/mem 到/从 reg: 1100010dw|mod|regr/m|
immed 到 reg/mem: 1110001lw|mod|regr/m|--data-- data 如 w=11
immed 到 register: 11011wreg|--data-- data 如 w=11
mem 到 accumulator: 11010000w|addr-low|addr-high|
accumulator 到 mem: 11010001w|addr-low|addr-high|
reg/mem 到 seg reg: 110001110|mod|0sgr/m| (sg=seg reg)
seg reg 到 reg/mem: 110001100|mod|0sgr/m| (sg=seg reg)
```

26.4.52 MOVS/MOVSX/MOVSZ/MOVSQ: 传送串

在存储单元之间传送数据。通常和 REP 前缀与 CX 中的长度一起使用, MOVSX 传送字节, MOVSZ 传送字, 而 MOVSQ(80386+)传送双字。操作数 1 按 ES:DI 寻址, 而操作数 2 按 DS:SI 寻址。如果方向标志是 0, 则该操作从左到右传送数据到操作数 1 的目的中, 同时 DI 和 SI 加 1、2 或 4。如果 DF 是 1, 则该操作从右到左传送数据, 并使 DI 与 SI 减量。对于每次重复, REP 都使 CX 减 1。当 CX 被减成 0 时, 该操作结束。在结束时, DI 和 SI 前进了步超过了所传送的最后一个字节。

标志: 不受影响。

源码: [REP] MOVSX/MOVSZ/MOVSQ(无操作数)

目标码: 1010010w

26.4.53 MOVSX/MOVZX: 带符号扩展或带零扩展的传送(80386+)

把 8 位或 16 位源操作数复制到较大的 16 位或 32 位目的操作数。MOVSX 把符号位填充到最左边的各个位, 而 MOVZX 则填充零。

标志: 不受影响。

源码: MOVSX/MOVZX register/memory, register/memory/immediate

目标码: MOVSX: 100001111 1011111w|modreg|r/m

MOVZX: 100001111 1011011w|modreg|r/m

26.4.54 MUL: 无符号数乘法

无符号的被乘数与无符号的乘数相乘。MUL 把最左边的 1 的位看作是数据位, 不是负的符号。该操作假定被乘数是在 AL、AX 或 EAX 中, 并根据乘数的大小决定其大小 (参见 IMUL)。下面是取决于乘数大小的乘法操作:

大小	被乘数	乘数	乘积	举例
8 位	AL	8 位 寄存器/存储器	AX	MUL BL
16 位	AX	16 位 寄存器/存储器	DX: AX	MUL mem-word
32 位	EAX	32 位 寄存器/存储器	EDX: EAX	MUL ECX

标志: 影响 CF 和 OF(AF, PF, SF 和 ZF 未定义)。

源码: MUL register/memory

目标码: 11111011w|mod10|r/m

26.4.55 NEG: 求补

将二进制值从正到负反向或从负到正反向。NEG 提供指定操作数的二进制补码, 采用从零中减操作数再加 1 的方法。操作数可以是在寄存器或存储器中的字节、字或双字(参见 NOT)。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: NEG register/memory

目标码: 11111011w|mod01|r/m|

26.4.56 NOP: 空操作

用于删除或插入机器码, 或为了定时目的的延迟执行。NOP 简单地用执行 XCHG AX, AX 的办法实现一次空操作。

标志: 不受影响。

源码: NOP (无操作数)

目标码: 10010000

26.4.57 NOT: 逻辑非

把 0 的位变成 1, 并把 1 的位变成 0。操作数是在寄存器或存储器中的字节、字或双字 (参见 NEG)。

标志: 不受影响。

源码: NOT register/memory

目标码: `·1111011w|mod010r/m|`

26.4.58 OR: 逻辑或

按 2 个操作数的位执行逻辑或操作。2 个操作数都是字节、字或双字, 其中位对位的相匹配。对于每个相匹配的位对, 如果二者中任何一个为 1 或二者都为 1, 则在第一个操作数中的该位被置 1; 否则, 该位不变(参见 AND 和 XOR)。

标志: 影响 CF(0), OF(0), PF, SF 和 ZF(AF 未定义)。

源码: OR register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 register: `1000010dw|modregr/m`

immed 到 accumulator: `10000110w|--data--|data 如 w=11`

immed 到 reg/mem: `1100000sw|mod001r/m|--data-- data 如 w=11`

26.4.59 OUT: 输出字节或字

从 AL 传送字节、从 AX 传送字或从 EAX 传送双字到输出端口。该端口是个固定的数值操作数或是在 DX 中的变量。如果端口号大于 256, 则使用 DX(参见 IN 和 OUTS)。

标志: 不受影响。

源码: 固定端口: OUT port#, AX

变量端口: OUT DX, AX

目标码: 固定端口: `11110011w|--port--|`

变量端口: `·1110111w|`

26.4.60 OUTS/OUTSB/OUTSW/OUTSD: 输出串(80286+)

把串(源)发送到端口。源是按 DS:SI 寻址的, DX 包含端口号。OUTS_n 通常是和 REP 前缀及包含被发送项(如字节、字或双字)数的 CX 一起使用的。根据方向标志(0/1), 该操作按项的大小使 SI 增量/减量(参见 IN 和 OUTS)。

标志: 不受影响。

源码: [REP] OUTSB/OUTSW/OUTSD(无操作数)

目标码: `0110111w`

26.4.61 POP: 字或双字出栈

把以前进栈的字或双字出栈到指定的目的地——存储单元, 通用寄存器或段寄存器。SP 指向在栈顶的当前字, POP 把它传送到指定目的地, 并使 SP 加 2。32 位操作数表示双字值, 并把 ESP 加 4(参见 PUSH)。

标志: 不受影响。

源码: POP register/memory

目标码: 3 种格式:

```
register:      101011regl
segment reg:  1000sg1111(sg 意思是 segment reg)
reg/memory:   1100011111mod000r/ml
```

26.4.62 POPA/POPAD: 所有通用寄存器出栈

POPA(80286+)使栈顶的 8 个字出栈到 DI, SI, BP, SP, BX, DX, CX 和 AX(按次序)。POPAD(80386+)使栈顶的 8 个双字出栈到 EDI, ESI, EBP, ESP, EBX, EDX, ECX 和 EAX。SP 的值被丢弃而不是被装入。通常, PUSHA/PUSHAD 以前已使寄存器进栈。

标志: 不受影响。

源码: POPA/POPAD(无操作数)

目标码: 01100001

26.4.63 POPF/POPFD: 标志出栈

POPF 使栈顶的字出栈到 16 位的标志寄存器中, 并使 SP 加 2。POPFD(80386+)使栈顶的双字出栈到 32 位的标志寄存器中, 并使 SP 加 4。通常 PUSHF 已经使标志进栈了。

标志: 影响所有标志。

源码: POPF/POPFD(无操作数)

目标码: 10011101

26.4.64 PUSH: 进栈

为后面的使用而使字或双字进栈。SP 指向栈顶的当前(双)字。PUSH 使 SP 减 2, 或使 ESP 减 4, 并从指定操作数传送一个(双)字到新的栈顶。源可以是通用寄存器, 段寄存器或存储器(参见 POP 和 PUSHF)。

标志: 不受影响。

源码: PUSH register/memory

PUSH immediate(80286+)

目标码: 3 种格式:

```

register:      |01010reg|
segment reg:  |000sg110| (sg 意思是 segment reg)
reg/memory:   |11111111|mod110r/m|

```

26.4.65 PUSHA/PUSHAD: 所有通用寄存器进栈

PUSHA(80286+)使 AX, CX, DX, BX, SP, BP, SI 和 DI 按次序进栈, 并使 SP 减 16。PUSHAD(80386+)使 EAX, ECX, EDX, EBX, ESP, EBP, ESI 和 EDI 进栈, 并使 SP 减 32。通常, POPA/POPAD 在后面使寄存器出栈。

标志: 不受影响。

源码: PUSHA/PUSHAD(无操作数)

目标码: 01100000

26.4.66 PUSHF/PUSHFD: 标志进栈

为后续使用而使标志寄存器的内容进栈。PUSHF 使 SP 减 2。PUSHFD(80386+)使 32 位标志寄存器进栈, 并使 SP 减 4(参见 POPF 和 PUSH)。

标志: 不受影响。

源码: PUSHF/PUSHFD(无操作数)

目标码: 10011100

26.4.67 RCL/RCR: 经过进位循环左移/经过进位循环右移

循环的位经过进位标志。该操作把在寄存器或存储器中的字节、字或双字循环左移或循环右移各个位。指定移位次数的操作数可以是立即数或是对 CL 的引用。在 8088/86 中, 常数只可能是 1, 较大的循环次数必须在 CL 中。在后继的处理器中, 该常数可以达到 31。对于 RCL, 最左边的位进入进位标志, 并且 CF 位进入目的的位 0, 所有其他位循环左移。对于 RCR, 位 0 进入进位标志, 并且 CF 位进入目的的最左边的位, 所有其他位循环右移(参见 ROL 和 ROR)。

标志: 影响 CF 和 OF。

源码: RCL/RCR register/memory, CL/immediate

目标码: RCL: |110100cw|mod010r/m| (如果 c=0, 移位次数是 1;
RCR: |110100cw|mod011r/m| 如果 c=1, 移位次数在 CL 中。)

26.4.68 REP: 重复串

把串操作重复指定次数。REP 是一个可选的重复前缀, 可以放在串指令 MOVS、STOS、INS 和 OUTS 的前面。在执行以前, 把一个计数值装入 CX。对于串指令的每次执行, REP 使 CX 减 1 并重复该操作, 直到 CX 为 0 为止, 在那一点上, 继续执行下一条指令(参见

REPE/REPZ/REPNE/REPNZ)。

标志：见相关的串指令。

源码：REP 串指令

目标码：11110010

26.4.69 REPE/REPZ/REPNE/REPNZ：有条件地重复串

把串操作重复指定次数或直到条件被满足为止。REPE、REPZ、REPNE 和 REPNZ 是可选的重复前缀，可以放在串指令 SCAS 和 CMPS 之前，这些串指令改变零标志。在执行以前，用一个计数值装入 CX 中。对于 REPE/REPZ(当相等/为零时重复)，该操作当 ZF 是 1(相等/为零的条件)且 CX 不是零时重复。对于 REPNE/REPNZ(当不相等/不为零时重复)，该操作是当 ZF 是 0(不相等/不为零的条件)且 CX 不是零时重复。当条件为真时，该操作使 CX 减 1 并执行串指令。

标志：见相关的串指令。

源码：REPE/REPZ/REPNE/REPNZ 串指令

目标码：REPNE/REPNZ: 11110010

REPE/REPZ: 11110011

26.4.70 RET/RETN/RETF：从过程返回

从以前用近 CALL 或远 CALL 进入的过程返回。汇编程序假定：如果 RET 是在标记为 NEAR 的过程之内，则为近 RET；如果是在标记为 FAR 的过程之内的，则为远 RET。对于近返回，RET 把栈顶的字传送到 IP 并使 SP 加 2。对于远返回，RET 把栈顶的字传送到 IP 和 CS 并使 SP 加 4。常数操作数(编码为 RET 4 的立即值)被加到 SP 上。

很明确，RETN 和 RETF 是用于编写近返回或远返回的。

标志：不受影响。

源码：RET/RETN/RETF[立即数]

目标码：4 种格式：

段内：1110000111

带立即数段内：1110000101data-low|data-high|

段间：1110010111

带立即数段间：1110010101data-low|data-high|

26.4.71 ROL/ROR：循环左移或循环右移

在寄存器或存储器中的字节、字或双字(80386+)循环左移或循环右移每个位。指定移位次数的操作数可以是立即数或是对 CL 的引用。在 8088/86 中，该常数只能是 1；较大的循环次数必须在 CL 中。在后继的处理器中，常数可以达到 31。对于 ROL，最左边的位进入目的位 0，所有其他位循环左移。对于 ROR，位 0 进入目的的最左边的位，所有其他位循环右

移(参见 RCL 和 RCR)。循环的位也进入进位标志。

标志: 影响 CF 和 OF。

源码: ROL/ROR register/memory, CL/immediate

目标码: ROL: 1110100cw|mod000r/m| (如果 c=0, 计数值=1;

ROR: 1110100cw|mod001r/m| 如果 c=1, 计数值在 CL 中。)

26.4.72 SAHF: AH 的内容存入标志

AH 的 8 位存入标志寄存器的低阶位(参见 LAHF)。

标志: 影响 AF, CF, PF, SF 和 ZF。

源码: SAHF(无操作数)

目标码: 10011110

26.4.73 SAL/SAR: 代数左移/代数右移

在寄存器或存储器中的字节、字或双字左移或右移各位。指定移位次数的操作数可以是立即常数或是对 CL 的引用。在 8088/86 中, 该常数只能是 1, 较大的移位次数必须在 CL 中。在后继的处理器中, 该常数可达 31。

SAL 左移指定数量的位, 并用 0 填充右边的空位。SAL 的动作恰好和 SHL 一样。SAR 是算术移位, 需要考虑被引用字段的符号。SAR 向右移指定数量的位并用符号位(0 或 1)填充到左边。所有移出的位都丢失了。

标志: 影响 CF, OF, PF, SF 和 ZF(AF 未定义)。

源码: SAL/SAR register/memory, CL/immediate

目标码: SAL: 1110100cw|mod100r/m| (如果 c=0, 计数值是 1;

SAR: 1110100cw|mod111r/m| 如果 c=1, 计数值在 CL 中。)

26.4.74 SBB: 带借位减法

常用于二进制多字减法, 把溢出的 1 的位带到算术运算的下一级。SBB 首先从操作数 1 中减去 CF 的内容(0/1), 然后就像 SUB 一样, 从操作数 1 中减去操作数 2 (参见 ADC)。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: SBB register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 reg: 1000110dw|modregr/m|

immed 从 accumulator: 10001110w|--data--|data 如 w=11

immed 从 reg/mem: 1100000sw|mod011r/m|--data--|data 如 sw=011

26.4.75 SCAS/SCASB/SCASW/SCASD: 扫描串

用于为指定的值扫描存储器中的串。对于 SCASB, 把该值装入 AL; 对于 SCASW, 把该值装入 AX; 对于 SCASD(80386+), 把该值装入 EAX。ES:DI 对引用被扫描的在存储器中的串。该操作通常是和 REPE/REPNE 前缀连同在 CX 中的计数值一起使用, 使用 REPE 是去找第一个不匹配的, 而使用 REPNE 则是去找第一个匹配的。如果方向标志是 0, 则该操作是从左到右扫描存储器, 并使 DI 增量。如果 DF=1, 则该操作是从右到左扫描存储器, 并使 DI 减量。REPn 对于每次操作都使 CX 减 1。该操作在相等(REPNE)或不相等(REPE)条件下或者当 CX 减到 0 时结束。最后的比较要清除或设置标志。如果指定的条件没有找到, REP 使 CX 减到 0; 否则, DI 和 SI 包含下一项的地址。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: [REPnn] SCASB/SCASW/SCASD(无操作数)

目标码: 1010111w

26.4.76 SETnn: 有条件地设置字节(80386+)

根据条件设置指定的字节。这是一个由 30 条指令组成的组, 包括 SET(N)E, SET(N)L, SET(N)C 以及 SET(N)S。这组指令恰好相当于条件转移(Jnn)的组。如果测试的条件为真, 则该操作把字节操作数设置为 1, 否则设置为 0。例如:

```
CMP    AX, BX           ; 比较 AX 和 BX 的内容
SETE   CL               ; 如果相等, 则设置 CL 为 1, 否则为 0
```

标志: 不受影响。

源码: SETnn register/memory

目标码: 100001111110C1cond!mod000r/m!
(cond 按照所测试的条件而改变)

26.4.77 SHL/SHR: 逻辑左移/逻辑右移

左移或右移寄存器或存储器中的字节、字或双字中的各位。指定移位次数的操作数可以是立即数或是对 CL 的引用。在 8088/86 中, 该常数只能是 1, 较大的移位次数必须在 CL 中。在后继的处理器中, 该常数可达 31。SHL 和 SHR 是把符号位看作数据位的逻辑移位。

SHL 向左移指定数量的位, 并把 0 填入右边腾空的位置。SHL 的动作和 SAL 一样。SHR 向右移指定数量的位并用 0 填充到左边。被移出的所有位都被丢弃了。

标志: 影响 CF, OF, PF, SF 和 ZF(AF 未定义)。

源码: SHL/SHR register/memory, CL/immediate

目标码: SHL: 1110100cw!mod100r/m! (如果 c=0, 计数值=1;

SHR: 1110100cw!mod101r/m! 如果 c=1, 计数值在 CL 中)

26.4.78 SHLD/SHRD: 双精度移位(80386+)

把多个位移入操作数中。该指令要求 3 个操作数。操作数 1 是包含被移位值的 16 或 32 位寄存器或存储单元。操作数 2 是包含要被移入操作数 1 的各个位的寄存器(和操作数 1 一样大小)。操作数 3 是包含移位值的 CL 或立即常数。

标志: 影响 CF, OF, PF, SF 和 ZF(AF 未定义)。

源码: SHLD/SHRD register/memory, register, CL/immediate

目标码: 00001111 10100100 modreg/mr

26.4.79 STC: 设置进位标志

设置 CF 为 1(见清除 CF 的 CLC)。

标志: 设置 CF。

源码: STC(无操作数)

目标码: 11111001

26.4.80 STD: 设置方向标志

设置 DF 为 1, 使串操作如 MOVS 从右到左进行处理(见清除 DF 的 CLD)。

标志: 设置 DF。

源码: STD(无操作数)

目标码: 11111101

26.4.81 STI: 设置中断标志

设置 IF 为 1, 在下一条指令执行之后, 可屏蔽外部中断(见清除 IF 的 CLI)。

标志: 设置 IF。

源码: STI(无操作数)

目标码: 11111011

26.4.82 STOS/STOSB/STOSW/STOSD: 存入串

把累加器的内容存入存储器。当和 REP 前缀连同 CX 中的计数值一起使用时, 该操作按指定次数复制串值, 适用于像清除一个存储区域那样的操作。STOSB 把值装入 AL, STOSW 把值装入 AX, 而 STOSD(80386+)把值装入 EAX。ES: DI 引用一个存储单元, 该值就被存放在那里。如果方向标志是 0, 则该操作从左到右存入存储器, 并使 DI 增量。如果 DF 是 1, 则该操作从右到左存入存储器, 并使 DI 减量, REP 对于每次重复都使 CX 减 1, 而当 CX 变成 0 时, 操作结束。

标志：不受影响。

源码：[REP] STOSB/STOSW/STOSD(无操作数)

目标码：1010101w

26.4.83 SUB：二进制值减法

从寄存器减去在寄存器、存储器或立即数中的二进制值，或者从存储器中减去在寄存器中的值或立即数。值可以是字节、字或双字(参见 SBB)。

标志：影响 AF, CF, OF, PF, SF 和 ZF。

源码：SUB register/memory, register/memory/ immediate

目标码：3 种格式：

reg/mem 与 register: 1001010dw|modregr/m:

immed 从 accumulator: 10010110w|--data--|data 如 w=11

immed 从 reg/mem: 1100000sw|mod101r/m|--data--|data 如 sw=011

26.4.84 TEST：测试位

对于一个特定的位配置，用 AND 逻辑去测试一个字段，但不改变目的操作数。2 个操作数都是在寄存器或存储器中的字节、字或双字，第二个操作数可以是立即数。如果任何一对相匹配的位是 1，则该操作清除 ZF，否则将 ZF 置 1。执行完之后，可以使用 Jnn 指令去测试标志。

标志：清除 CF 和 OF 标志，并影响 PF, SF 和 ZF。(AF 未定义)。

源码：TEST register/memory, register/memory/immediate

目标码：3 种格式：

reg/mem 与 register: 11000010w|modregr/m:

immed 到 accumulator: 11010100w|--data-- data 如 w=11

immed 到 reg/mem: 11111011w|mod000r/m|--data--|data 如 w=11

26.4.85 WAIT：置处理器于等待状态

允许处理器保持在等待状态直到发生一个外部中断为止，目的是为了和协处理器同步。处理器等待直到协处理器完成执行，并在 TEST 引线接收到信号时立即恢复处理。

标志：不受影响。

源码：WAIT(无操作数)

目标码：10011011

26.4.86 XADD：交换并相加(80486+)

把源操作数和目的操作数相加，和存入目的，并把原先的目的的值传送到源。

标志: 影响 AF, CF, OF, PF, SF 和 ZF。

源码: XADD register/memory, register

目标码: 10000111111100000b|modreg/m|

26.4.87 XCHG: 交换

在 2 个寄存器之间交换数据(如 XCHG AH, BL)或在寄存器与存储器之间交换数据(如 XCHG CX, word)。

标志: 不受影响。

源码: XCHG register/memory, register/memory

目标码: 2 种格式:

reg 与 accumulator: 110010reg|

reg/mem 与 reg: 11000011w|modreg/m|

26.4.88 XLAT/XLATB: 换码

把字节转换成不同格式, 比如加密的数据。把转换表的地址装入 BX 或对 EBX(对于 32 位大小), 然后把被转换的值装入 AL。该操作把 AL 的值作为偏移值进入表中, 从表中选择字节, 并存入 AL(XLATB 是 XLAT 的同义词)。

标志: 不受影响。

源码: XLAT/XLATB [AL](AL 操作数是可选的)

目标码: 11010111

26.4.89 XOR: 异或

实现 2 个操作数的按位逻辑异或。2 个操作数都是字节、字或双字, 在其中 XOR 按位对位相匹配。对于每对相匹配的位, 如果 2 个位都相同, 则第一个操作数中的对应位被清除为 0; 如果相匹配的位是不同的, 则在第一个操作数中的对应位被置 1(参见 AND 和 OR)。

标志: 影响 CF(0), OF(0), PF, SF 和 ZF(AF 未定义)。

源码: XOR register/memory, register/memory/immediate

目标码: 3 种格式:

reg/mem 与 reg: 1001100dw|modreg/m|

immed 到 reg/mem: 11000000w|mod110r/m|--data-- data 如 w=1|

immed 到 accumulator: 10011010w|--data--|data 如 w=1.

A.2 十进制数转换成十六进制数

为了把十进制数 42 936 转换成十六进制数，首先将 42 936 除以 16，余数成为最右边的十六进制数位 8。接着，新的商 2 683 除以 16，余数是 11 即 B，成为左边的下一个十六进制数位。用这种方法继续从每一步除法的余数中求出十六进制数，直到商是 0 为止。这些步骤是如下进行的：

操作	商	余数	十六进制
42936/16	2683	8	8 (最右边的位)
2683/16	167	11	B
167/16	10	7	7
10/16	0	10	A (最左边的位)

也可以使用表 A-1 进行十进制到十六进制的转换。对于十进制数 42 936，定位在表中的数，该数是等于或下一个比它小的数。注意等效的十六进制值和它在表中的位置。从 42 936 中减去该十六进制数位的十进制值，并且定位在表中的差值。该过程如下进行：

	十进制	十六进制
开始的十进制值	42936	
减去下一个较小的数	<u>-40960</u>	A000
差值	1976	
减去下一个较小的数	<u>-1792</u>	700
差值	184	
减去下一个较小的数	<u>-176</u>	B0
差值	8	8
最终的十六进制值		<u>A7B8</u>

十六进制-十进制转换表

表 A-1

H e x	Dec	H e x	Dec	H e x	Dec	H e x	Dec	H e x	Dec	H e x	Dec	H e x	Dec	H e x	Dec
0	268,435,456	0	16,777,216	0	1,048,576	0	65,536	0	4,096	0	256	0	16	0	0
1	536,870,912	1	33,554,432	1	2,097,152	1	131,072	1	8,192	1	512	1	32	1	1
2	805,306,368	2	50,331,648	2	4,194,304	2	262,144	2	16,384	2	1,024	2	64	2	2
3	1,073,741,824	3	67,108,864	3	8,388,608	3	524,288	3	32,768	3	2,048	3	128	3	3
4	1,342,177,280	4	83,886,080	4	10,582,912	4	686,432	4	43,008	4	2,752	4	176	4	4
5	1,610,612,736	5	100,663,296	5	12,777,344	5	848,896	5	53,760	5	3,408	5	224	5	5
6	1,879,048,192	6	117,440,512	6	14,971,776	6	1,010,304	6	64,512	6	4,096	6	280	6	6
7	2,147,483,648	7	134,217,728	7	17,166,208	7	1,272,704	7	80,256	7	5,120	7	336	7	7
8	2,415,919,104	8	150,994,144	8	19,360,640	8	1,535,168	8	96,000	8	6,144	8	392	8	8
9	2,684,354,560	9	167,772,160	9	21,555,072	9	1,797,600	9	112,704	9	7,168	9	448	9	9
A	2,952,790,016	A	184,549,376	A	23,749,504	A	2,058,048	A	128,448	A	8,192	A	504	A	10
B	3,221,225,472	B	201,326,592	B	25,943,936	B	2,318,496	B	144,192	B	9,216	B	560	B	11
C	3,489,660,928	C	218,103,808	C	28,138,368	C	2,578,944	C	160,432	C	10,240	C	616	C	12
D	3,758,096,384	D	234,881,024	D	30,332,800	D	2,839,392	D	176,176	D	11,264	D	672	D	13
E	4,026,531,840	E	251,658,240	E	32,527,232	E	3,099,840	E	192,416	E	12,288	E	728	E	14
F		F		F		F		F		F		F		F	15

ASCII 字符码

术语 ASCII 代表 “American Standard Code for Information Interchange” (美国信息交换标准码)。表 B-1 列出了完整的 256 个 ASCII 字符码 (00H 到 FFH)，以及对应的十六进制表示。字符码的种类是：

00-1F 有关屏幕、打印机和数据传送的控制码，它们是用来产生动作的。

20-7F 有关数字、字母和标点符号的字符码(20H 是标准的空格或空白)。

80-FF 扩展的 ASCII 码，外来字符，希腊字母和数学符号，以及画框图用的图形符号。

下面是控制码 00H 到 1FH 及其说明：

十六进制	字符	十六进制	字符	十六进制	字符
00	空	01	标题开始	02	正文开始
03	正文结束	04	传输结束	05	询问
06	确认	07	响铃	08	退格
09	水平制表	0A	换行	0B	垂直制表
0C	换页	0D	回车	0E	移出
0F	移入	10	数据行换码	11	设备控制 1
12	设备控制 2	13	设备控制 3	14	设备控制 4
15	不确认	16	同步空闲	17	信息块传输结束
18	取消	19	媒介结束	1A	置换
1B	换码	1C	文件分隔符	1D	组分分隔符
1E	记录分隔符	1F	单元分隔符		

表 B-1

ASCII 字符集

00		20		40	@	60	`	80	Ç	A0	À	C0	Ì	E0	α
01	⊙	21	!	41	A	61	a	81	ú	A1	Á	C1	Í	E1	β
02	●	22	"	42	B	62	b	82	ê	A2	Â	C2	Î	E2	Γ
03	♥	23	#	43	C	63	c	83	ë	A3	Ã	C3	Ï	E3	π
04	♦	24	\$	44	D	64	d	84	ä	A4	Ä	C4	Ñ	E4	Σ
05	♣	25	%	45	E	65	e	85	å	A5	Å	C5	Ò	E5	σ
06	♠	26	&	46	F	66	f	86	ä	A6	•	C6	Ó	E6	μ
07		27	'	47	G	67	g	87	ç	A7	°	C7	Ô	E7	τ
08		28	(48	H	68	h	88	è	A8	¿	C8	Õ	E8	φ
09		29)	49	I	69	i	89	é	A9	¬	C9	Ö	E9	θ
0A	*	2A	*	4A	J	6A	j	8A	ê	AA	¸	CA	×	EA	Ω
0B	+	2B	+	4B	K	6B	k	8B	ï	AB	½	CB	÷	EB	δ
0C	2C	2C	,	4C	L	6C	l	8C	î	AC	¼	CC	—	EC	ω
0D	2D	2D	-	4D	M	6D	m	8D	í	AD	¼	CD	=	ED	φ
0E	2E	2E	.	4E	N	6E	n	8E	Ä	AE	½	CE	+	EE	ε
0F	2F	2F	/	4F	O	6F	o	8F	Å	AF	¾	CF	—	EF	∩
10	►	30	0	50	P	70	p	90	É	B0	¾	D0	—	F0	∞
11	◄	31	1	51	Q	71	q	91	æ	B1	¾	D1	—	F1	±
12	+	32	2	52	R	72	r	92	æ	B2	¾	D2	—	F2	±
13		33	3	53	S	73	s	93	ö	B3	¾	D3	—	F3	±
14	¶	34	4	54	T	74	t	94	ö	B4	¾	D4	—	F4	±
15	§	35	5	55	U	75	u	95	ö	B5	¾	D5	—	F5	±
16	—	36	6	56	V	76	v	96	ü	B6	¾	D6	—	F6	±
17	±	37	7	57	W	77	w	97	ü	B7	¾	D7	—	F7	±
18	↑	38	8	58	X	78	x	98	ÿ	B8	¾	D8	—	F8	±
19	↓	39	9	59	Y	79	y	99	ÿ	B9	¾	D9	—	F9	±
1A		3A	:	5A	Z	7A	z	9A	ÿ	BA	¾	DA	—	FA	±
1B		3B	;	5B	[7B	{	9B	ÿ	BB	¾	DB	—	FB	±
1C	⌞	3C	<	5C	\	7C	}	9C	ÿ	BC	¾	DC	—	FC	±
1D	↔	3D	=	5D]	7D	~	9D	ÿ	BD	¾	DD	—	FD	±
1E	▲	3E	>	5E	^	7E	~	9E	ÿ	BE	¾	DE	—	FE	±
1F		3F	?	5F	_	7F	Δ	9F	f	BF	¾	DF	—	FF	±

DOS DEBUG 程序对于编写很小的程序，调试汇编程序以及检查文件或存储器的内容是有用的。你可以在 DOS 下面名为 \DOS 的目录中，或在 Windows95/98 下面通过开始菜单选择 MS-DOS 提示符，找到 DEBUG.EXE。为了打印显示的副本，简便的方法是在窗口中运行 DEBUG。使用鼠标指定一个区域，将其复制到剪贴板，接着就可以把它粘贴到记事本或字处理程序中。

为了开始该程序，键入 DEBUG 并按回车键，DEBUG 会从磁盘装入到存储器中。当 DEBUG 的提示符短划(-)出现在屏幕上时，DEBUG 已准备好接受命令(那是一个短划，尽管它和光标很相像)。

可以键入一条或两条命令启动 DEBUG：

1. 为建立文件或检查存储器，键入不带文件说明的 DEBUG；
2. 为了修改或调试程序(.COM 或.EXE)或者修改文件，键入带文件说明的 DEBUG，比如 DEBUG n:PROGC.COM。

装入程序把 DEBUG 装入到存储器中，并且 DEBUG 显示一个短划(-)作为提示符。用 256 个字节(100H)的程序段前缀(PSP)地址初始化 CS，DS，ES，以及 SS 寄存器，同时工作区从 PSP+100H 处开始。下面是 DEBUG 如何从左到右显示标志寄存器的：

	溢出	方向	中断	符号	零	辅助进位	奇偶	进位
为 1	OV	DN	EI	NG	ZR	AC	PE	CY
为 0	NV	UP	DI	FI	NZ	NA	PO	NC

可以根据段：偏移值(如 DS:120)引用存储器地址，或者只根据偏移值(如 120)引用。也可以直接引用存储器地址，如 40:17，其中 40[0]H 是段地址，而 17H 是偏移值。注意，.EXE 程序的数据段是从 DS:0 开始的，而.COM 程序则是从 DS:100 开始的。

DEBUG 假定所有输入的数都是十六进制的，所以不用键入后面的 H。F1 和 F3 键是以如下方式工作的：F1 每次按键重复前一个命令，而 F3 键却重复前面的全部命令。另外，DEBUG 不区分大、小写字母。

以下是按字母顺序对每个 DEBUG 命令的说明。

A(汇编)。把汇编源语句转换成机器码。这个操作对于编写和测试小的汇编程序，以及检查小的代码段特别有用。代码的默认起始地址是 CS:0100H。A 命令的格式是

A [address]，其中 address 默认为 100H。

DEBUG 支持以下特性：

- DB 和 DW 可以用于定义程序需要引用的数据项。
- 操作数在方括号内, 比如[12E], 它指定存储器偏移值。
- PTR 操作符可以用于指定字节或字, 如 INC BYTE PTR [12E]。
- 寄存器-间接操作数的所有形式都是适用的, 比如[BP+DI]和 25[BX]。

下面的例子建立一个包含 5 个语句的汇编程序。键入指令, DEBUG 产生代码段(这里表示为 xxxx:)和从 0100H 开始的偏移值

	A(或 A 100) <Enter>	说明
xxxx: 0100	MOV CX, [10D] <Enter>	; 取 10D 处的内容
xxxx: 0104	ADD CX, 1A <Enter>	; 加立即值
xxxx: 0107	MOV [10D], CX <Enter>	; 把 CX 存入 10D 中
xxxx: 010E	JMP 100 <Enter>	; 转移返回到起点
xxxx: 010D	DW 2500 <Enter>	; 定义常数
	<Enter>	; 命令结束

根据 PSP 的大小, DEBUG 把 IP 设置为 100H, 这样语句就从 100H 开始。最后的<Enter>(回车键)(在一行中是 2 个)通知 DEBUG 结束该程序。现在可以使用 U(反汇编)命令检查机器码并使用 T(跟踪)命令跟踪程序的执行。

改变上述的任一条指令或数据项, 所提供的新长度和老长度是一样的。例如, 把在 104H 的 ADD 改变为 SUB, 键入

```
A 104 <Enter>
xxxx: 104 SUB CX, 1A <Enter> <Enter>
```

重新执行该程序时, IP 仍然增量。使用寄存器(R)命令把它复位成 100H。使用 Q 退出。

C(比较)。比较存储器的 2 个区的内容, 默认寄存器是 DS。可以编写带有长度或范围的命令:

1. C *start-from-addr length start-to-addr*。下面的例子把从 DS:050 开始的 20H 字节与从 DS:200 开始的字节进行比较:

```
C 050 120 200 ; 使用长度 20H 的比较
```

2. C *start-from-addr end-from-addr start-to-addr*。这一例子是把从 DS:050 开始到 DS:70 的字节与从 DS:200 开始的字节进行比较:

```
C 050 070 200 ; 使用范围的比较
```

该操作显示不相等字节的地址和内容。

D(显示或转储)。以十六进制和 ASCII 形式显示部分存储器的内容, 默认的寄存器是 DS。可以编写带有长度或范围的命令:

1. D [*start-address* [*length*]]。指定带有可选长度的起始地址, 省略长度时的默认值为 80H(128 字节)。下面是一些例子:

```
D 200 ; 显示从 DS:200H 处开始的 80H 个字节
D ; 显示从上一次显示结束处开始的 80H 个字节
D CS:150 ; 显示从 CS:150H 开始的 80H 个字节
D DS:20 L5 ; 显示从 DS:20H 开始的 5 个字节
```

2. D [*start-addr end-addr*]。如下所示:

D 300 32C ; 显示从 300H 到 32CH 的字节

E(写入)。允许键入数据或机器指令，默认寄存器是 DS，格式是 E *address* [*list*]。

该操作有 2 个选项：

1. 用列表(list)中的字节替换字节，如下所示：

```
E 105 13 3A 21 ; 输入从 DS:105H 开始的 3 个字节
E CS:211 21 2A ; 输入从 CS:211H 开始的 2 个字节
E 110 'anything' ; 输入从 DS: 110H 开始的字符串
```

字符串要使用单引号或双引号。

2. 允许顺序地编辑字节；键入想要显示的地址 (*address*)：

E 12C ; 显示 DS:12CH 的内容

该操作等待从键盘的输入。从 DS:12CH 开始键入一个或多个十六进制值的字节，用空格隔开。

F(填充)。用列表中的值填充某个范围内的存储单元，默认寄存器是 DS。可以编写带长度或范围的命令：

1. F *start-addr* *length* '*data*'。下面是一个例子：

F 210 L19 'Help!' ; 使用 19H(25)个字节的长度

2. F *start-addr* *end-addr* '*data*'。下面是一个例子：

F 210 229 'Help!' ; 使用范围，210H 到 229H

2 个例子都用包含 'Help!' 的字节填充从 DS:210H 开始的存储器单元。

G(运行)。执行正在调试的机器语言程序一直到达指定断点。要保证检查所列出的机器码是有效的 IP 地址，因为无效的地址可能导致无法预料的后果。另外，只能在你自己的程序中设置断点，不能在 DOS 或 BIOS 的程序模块中设置。该操作通过 INT 操作执行并暂停，如有必要，还要等待键盘输入。默认的寄存器是 CS，格式是：

G [-*start-address*] *break-address* [*break-address*...]

项目 =*start-address* 是可选的。其他项目允许多达 10 个断点的地址。命令 G 11A 通知 DEBUG 从 IP 的当前单元开始执行所有指令，直到单元 11A 为止。

H(十六进制)。显示 2 个十六进制值的和与差，编码为 H *value value*。最大长度是 4 个十六进制数位。例如，命令 H 14F 22 显示结果 171(和)与 12D(差)。

I(输入)。输入并显示来自端口的一个字节，编码为 I *port-address*。

L(装入)。把一个文件或磁盘扇区装入存储器。文件可以是被“命名”的，以便 DEBUG 用下面的两种方法之一识别它：DEBUG 要求带文件说明执行，或者在 DEBUG 内执行 N(命名)命令。L 命令有 2 种格式：

1. 装入一个命名的文件：L [*address*]。使用地址参数使 L 从指定单元开始进行装入。省略地址，会使 L 从 CS:100 处进行装入。要想装入没有命名的文件，应当首先给它命名(参见 N 命令)：

```
N filespec ; 命名文件
L ; 在 CS:100 处装入该文件
```

为了重新装入该文件，只需发出不带地址的 L 命令，DEBUG 重新装入文件并相应地初始化寄存器。

2. 装入来自磁盘扇区的数据：L [address [drive start number]]。
 - Address 为装入数据提供起始的存储单元(默认值是 CS:100)。
 - Drive 指明磁盘驱动器，其中 0=A, 1=B, 等等。
 - Start 指定要装入的第一个扇区的十六进制编号(相对数，其中柱面 0、磁道 0、扇区 1，是相对扇区 0 而言的)。
 - Number 给出进行装入的顺序扇区的十六进制数。

下面的例子是准驱动器 0(A)的从扇区 20H 开始的 15 个扇区装入到以 CS:100 为起点的存储区：

```
T, 100 0 20 15
```

L 操作把装入的字节数回送到 BX: CX。对于 EXE 文件，DEBUG 忽略地址参数(如果有的话)而使用 EXE 标题中的装入地址。它还会去掉标题，为了保存标题，要在执行 DEBUG 前，用不同的扩展名重新命名该文件。

M(传送)。传送(或复制)存储单元的内容，默认的寄存器是 DS。可以编写带长度或范围的命令：

1. M start-addr length end-addr。一个例子：

```
M DS: 50 100 DS:400 ; 使用传送的长度
```

2. M start-from-addr end-from-addr start to-addr。一个例子：

```
M DS:50 150 DS:400 ; 使用传送的范围
```

2 个例子都把从 DS:050H 开始到 150H 的字节复制到从 DS:400H 开始的地址。

N(命名)。命名从磁盘上读出或向磁盘写入的程序或文件。命令编码为 N filespec，比如

```
N path: SAM.COM
```

该操作把名字存入 PSP 中的 CS:80 处。CS:80 处的第一个字节标明长度(0AH)，接着是空格和文件说明。然后可以使用 L(装入)或 W(写)读或写该文件。

O(输出)。发送字节到某个端口，编码为 O port-address byte。

P(继续进行)。执行 CALL, LOOP, INT 或重复串指令(REP)直到下一条指令。它的格式是：P [=address] [value]，其中 =address 是可选的起始地址，value 是可选的继续进行下去的指令数。省略 =address 默认为当前的 CS:IP 值。例如，跟踪执行的是 INT 10H 操作，那么只要键入 P 就可以从头到尾地执行整个操作。参见 G 和 T。

Q(退出)。退出 DEBUG。该操作不保存文件，为了保存文件，可使用 W。

R(寄存器)。显示寄存器的内容和下一条指令。它的格式是 R [register-name]。以下例子说明这一命令的用法：

```
R      显示所有寄存器
```

```
R DX   显示 DX，DEBUG 提供如下选项：
```

(1)按回车键，保持 DX 不变；

(2)键入 1 到 4 个十六进制数位改变 DX 的内容。

R IP 显示 IP。键入另一个值改变 IP 的内容。

R F 显示标志的当前设置，每个标志表示为 2 个字母代码。见本附录中有关标志名的说明。

S(查找)。为列表中的字符查找存储器。如果字符被找到，该操作返回它们的地址，否则，不响应。默认的寄存器是 DS。可以编写带长度或范围的命令：

1. `S start-addr length 'data'`。下面的例子是在开始于 DS:300，长度为 2000H 字节的范围内查找“VIRUS”：

```
S 300 L2000 "VIRUS"
```

2. `S start-from-addr end-from-addr start-to-addr`。下面的例子是从 CS:100 到 CS:400 的范围内查找内容为 51H 的字节：

```
S CS: 100 400 51
```

T(跟踪)。以单步方式执行程序。注意，通常应当使用 P 执行整个 INT 指令。默认的寄存器是 CS:IP，格式是 `T [=address] [value]`。可选的项目 `=address` 通知 DEBUG 从哪里开始跟踪，而可选的 `value` 则给出跟踪的指令数。省略这两个操作数，DEBUG 执行下一条指令并显示寄存器。2 个例子：

```
T           ; 执行下一条指令
T 10        ; 执行下面 10H(16)条指令
```

T 命令试图执行所有指令，无效的指令会导致处理器死锁，需要重新引导才行。

U(反汇编)。将机器指令进行反汇编，即把机器指令转换为符号码。默认的寄存器是 CS:IP，它的格式是

```
U [start-addr]或 U [start-addr end-addr]
```

所指定的区域应当包含有效的机器码，该操作显示的是符号指令。3 个例子：

```
U 100       ; 反汇编开始于 CS:100 的 32 个字节
U           ; 反汇编上一个 U 以后的 32 个字节(如果有的话)
U 100 140    ; 反汇编从 100H 到 140H 的内容
```

DEBUG 不能正确地转换某些条件转移和专门用于 80 286 及其后继处理器的指令，尽管它能正确地执行(DEBUG 将它们表示为 DB 语句)。

W(写)。从 DEBUG 写入文件。如果文件还没有被装入，那么它应当首先被命名(参见 N)。默认的寄存器是 CS，它的格式是

```
W [address[drive start-sector number-of-sectors]]
```

只能写入带.COM 扩展名的程序文件，因为 W 不支持.EXE 格式(为了修改.EXE 程序，可临时改变扩展名)。可以使用 DEBUG，在两种情况下，向磁盘上写入程序：

1. 从磁盘上检索一个已存的程序，修改然后保存它，需要遵循以下步骤：
 - 使用机器语言程序的名字将其装入存储器：`DEBUG n:filename.ext`。
 - 使用 D 命令观察该程序并使用 E 命令进行修改。
 - 使用 W 命令写入修改过的程序。
2. 使用 DEBUG 建立一个非常小的机器语言程序，若想保存它，需要遵循以下步骤：

- 使用 A(汇编)和 E 命令键入源程序。
- 输入 N filename.COM 为程序命名, 该程序的扩展名必须是.COM。
- 由于需要知道程序实际在哪里结束, 所以要以字节计的程序的大小插入 BX:CX 对。考虑这一程序的例子是:

```
xxxx: 0100 MOV CL, 42
xxxx: 0102 MOV DL, 2A
xxxx: 0104 ADD CL, DL
xxxx: 0106 JMP 100
```

虽然键入符号码, 但 DEBUG 还要产生机器码, 这就是要进行保存的。因为最后一条指令 JMP 是两个字节, 所以该程序大小是从 100H 到 107H, 即 8。

- 首先使用 R BX 显示 BX(大小的高阶部分), 并输入 0 清除它。
- 接着使用 R CX 显示 CX。DEBUG 用 CX nnnn 回应(它可能包含任何值), 这时应该用程序大小 8 来替代它。
- 键入 W <Enter>, 在磁盘上写入修改过的程序。

DEBUG 显示信息 “Writing 8 bytes”(“写 8 个字节”)。如果这个数字是零, 可能在输入程序长度时有失误, 要再试一次。要特别注意程序的大小, 该值是十六进制的, 并且最后一条指令的长度可能会超过一个字节。



保留字

汇编程序认定某些具有特定意义的字，只有在规定条件下，才可以使用。汇编程序保留的字可以划分为4种类型：

- (1) 寄存器名，如 AX 和 AH。
- (2) 符号指令，如 ADD 和 MOV。
- (3) 伪操作(给汇编程序的命令)，如 PROC 和 END。
- (4) 操作符，如 DUP 和 SEG。

如果用下面的保留字定义数据项，会使汇编程序混淆或造成汇编错误。个别的汇编程序版本除这里所列出的之外，还可能有其他保留字。

寄存器名

AH, AL, AX, BH, BL, BP, BX, CH, CL, CS, CX, DH, DI, DL, DS, DX, EAX, EBP, EBX, ECX, EDI, EDX, EIP, ES, ESI, FS, GS, IP, SI, SP, SS。

符号指令

AAA, AAD, AAM, AAS, ADC, ADD, AND, ARPL, BOUND, BSF, BSR, BT_n, CALL, CBW, CDQ, CLC, CLD, CLI, CLTS, CMC, CMP, CMPS_n, CMPXCHG, CMPXCHG8B, CWD_n, DAA, DAS, DEC, DIV, ENTER, ESC, HLT, IDIV, IMUL, IN, INC, INS_n, INT, INTO, IRET, JA, JAE, JB, JBE, JCXZ, JE, JECXZ, JG, JGE, JL, JLE, JMP, JNA, JNAE, JNB, JNBE, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LAHF, LAR, LDS, LEA, LEAVE, LES, LFS, LGDT, LGS, LIDT, LLDT, LMSW, LOCK, LODS_n, LOOP, LOOPE, LOOPNE_n, LOOPNZ_n, LOOPZ, LSL, LSS, LTR, MOV, MOVS_n, MOVSB, MOVSD, MOVZX, MUL, NEG, NOP, NOT, OR, OUT_n, POP, POPA, POPAD, POPF, POPFD, PUSH, PUSHAD, PUSHF, PUSHFD, RCL, RCR, REN, REP, REPE, REPNE, REPZ, RET, RETF, ROL, ROR, SAHF, SAL, SAR, SBB, SCAS_n, SET_{mn}, SGDT, SHL, SHLD, SHR, SHRD, SIDT, SLDT, SMSW, STC, STD, STI, STOS_n, STR, SUB, TEST, VERR, VERRW, WAIT, XADD, XCHG, XLAT, XOR。

伪操作

ALIGN, .ALPHA, ASSUME, BYTE, .CODE, COMM, COMMENT, .CONST, .CREF,

.DATA, .DATA?, DB, DD, DF, DOSSEG, DQ, DT, DW, DWORD, ELSE, END, ENDIF, ENDM, ENDP, ENDS, EQU, .ERRnn, EVEN, EXITM, EXTRN, EXTERN, .FARDATA, .FARDATA?, FWORD, GROUP, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFE, IFIDN, IFNB, IFNDEF, INCLUDE, INCLUDELIB, IRP, IRPC, LABEL, .LALL, .LFCOND, .LIST, LOCAL, MACRO, .MODEL, NAME, ORG, &OUT, PAGE, PROC, PUBLIC, PURGE, QWORD, .RADIX, RECORD, REPT, .SALL, SEGMENT, .SEQ, .SFCOND, .STACK, STRUC, SUBTTL, .TFCOND, TITLE, TWORD, UNION, WORD, .XALL, .XCREF, .XLIST。

操作符

AND, BYTE, COMMENT, CON, DUP, EQ, FAR, GE, GT, HIGH, LE, LENGTH, LINE, LOW, LT, MASK, MOD, NE, NEAR, NOT, NOTHING, OFFSET, OR, PTR, SEG, SHL, SHORT, SHR, SIZE, STACK, THIS, TYPE, WHILE, WIDTH, WORD, XOR。



本附录涉及汇编, 连接, 产生交叉引用文件, 以及把.EXE 程序转换为.COM 格式的一些规则。把源程序转换为可执行文件的步骤是:

1. 汇编源程序。这一步建立目标(.OBJ)文件, 可选的列表(.LST)文件, 以及可选的用作交叉引用的文件。

2. 连接目标文件。这一步建立可执行(.EXE)文件, 可选的映像(.MAP)文件, 以及可选的库(.LIB)文件(连接可以把一个以上的目标文件组合成可执行文件)。

3. 可以有选择地把.EXE 文件转换成.COM 文件。

不同的汇编程序版本对这些步骤的处理是不一样的, 并且提供了很多的选项序列, 这里所涉及的只是一部分。本附录包括 Microsoft MASM 6.1, MASM 5.1, 以及 Borland Turbo TASM 的版本。

用在本附录中的路径对于程序以及文件来说意思是文件和目录, 可以把它替换为适当的值, 比如 C:或 C:\子目录。

E.1 MICROSOFT MASM 6.1

Microsoft MASM 6.1 汇编程序使用 ML 命令, 但为了和较早的版本兼容也接受 MASM 命令。ML 命令允许把任意多的程序汇编和连接成为一个可执行模块。用于汇编与连接的 ML 命令的格式是

```
ML [options] filename.ASM [[options] filename.ASM]...[/LlNK options]
```

命令行的选项(options)是大小写敏感的, 由斜杠(/)字符开始。包括以下这些:

/AT 指明程序使用 Tiny 存储模型, 该命令把程序转换成.COM 格式。

/c 只汇编, 不连接。

/F1 产生列表(.LST)文件。

/Fm 产生连接映像(.MAP)文件。

/Fr 产生.SBR(交叉引用)文件。

/Sn 禁止符号表的列表。

/Zd 包含调试信息的行号。

/Zi 包含符号的调试信息(对于 Code View)

两个例子:

```
ML /Sn /Fr program23.asm {汇编并连接}
ML /AT /Zd program34.asm {汇编, 连接, 产生.COM文件}
```

一个有用的命令是 **ML-?**, 它显示完整的命令行语法和选项。

也可以使用 **MASM** 命令行, 它起的作用和 **MASM 5.1** 一样 (见下一节 **MASM 5.1**)。

.COM 程序。如果源程序使用 Tiny 存储模型, 则 **ML** 命令产生 **.COM** 程序。如果程序使用常规段, 则 **ML** 产生 **.EXE** 文件, 这时必须使用 **EXE2BIN** 把它转换成 **.COM** 格式 (见下一节 **MASM 5.1**)。

连接。可以只汇编而不连接 (使用 **/c** 选项), 汇编并连接 (默认的), 在命令行中指定 **LINK**, 或者单独地运行 **LINK** 程序。**LINK** 的格式是

```
LINK [options] object-file(s), [exefile], [mapfile], [libraries]
```

为了显示全部连接选项, 可以键入 **LINK/?**。

交叉引用文件。使用 **ML** 命令中的 **/Fr** 选项产生 **.SBR** 文件, 使用 **CREF** 程序把它转换成特有的、已排序的 **.CRF** 文件:

```
CREF xreffile.SBR, reffile.REF
```

E.2 MICROSOFT MASM 5.1

这一版本要求采用分别的步骤对程序进行汇编, 连接, 以及转换成 **.COM** 格式。该汇编程序命令行的格式是

```
MASM [options] source-file[, object-file][, list-file][, xref-file]
```

- **Options** 以后说明。
- **Source-file** 指明源程序。汇编程序缺省的扩展名为 **.ASM**, 所以不需要输入它。可以键入路径, 如 **C:\子目录\文件名**。
- **Object-file** 用来产生 **.OBJ** 文件。
- **List-file** 用来产生包含源码与目标码的 **.LST** 文件。
- **Xref-file** 用于产生 **.CRF** 交叉引用文件, 该文件包含作为交叉引用列表的符号。

每个文件可以有它自己的路径与文件名, 文件名可以和源相同或不同。下面例子清楚说明了所有文件:

```
MASM prog-name, prog-name, prog-name, prog-name
```

下面的简化命令允许默认目标文件和交叉引用文件, 二者都具有相同的名字, 但是没有列表文件: **MASM filename,...**。 **MASM** 包括以下一些选项(options):

- /C** 在 **.LST** 文件中建立交叉引用表。
- /L** 建立标准的列表(**.LST**)文件。该命令行还为这一选项提供路径。
- /N** 在 **.LST** 文件中禁止符号表的产生。
- /Z** 出错时在屏幕上显示源行。

/ZD 为 Code View 在目标文件中包含行号信息。

/ZI 为 Code View 在目标文件中包含行号和符号的信息。

以下例子需要两个选项: **MASM /L/Z** 文件名。为了简要地说明所有选项, 不带文件名或其他选项键入 **MASM /H**(帮助)。

交叉引用文件。CRF 文件用于产生程序的标号, 符号和变量的列表。使用 **CREF** 把 CRF 文件转换成排序的交叉引用(.REF)文件:

```
CREF xreffile, reffile
```

- **xreffile** 指明由汇编程序所产生的交叉引用文件。该程序缺省了扩展名, 所以不需要输入它。可以指定路径。
- **reffile** 用于产生 REF 文件。路径与文件名可以和源的路径与文件名一样或不同。

下面的例子编写一个在驱动器 C 上的名为 **ASMPROG.REF** 的交叉引用文件:

CREF C:ASMPROG, C:。

连接。连接 **MASM 5.1** 程序的命令行是

```
LINK objfile, exefile[, mapfile[, libraryfile]
```

- **Objfile** 指明由汇编程序所产生的目标文件。连接程序缺省扩展名为 **.OBJ**, 所以不需要再输入它。路径与文件名可以和源中的路径与文件名一样或不同。
- **Exefile** 用于产生 **.EXE** 文件。路径与文件名可以和源中的路径与文件名一样或不同。
- **Mapfile** 用于产生 **.MAP** 文件, 该文件指明每一个段的相对位置与大小以及连接程序所发现的任何错误。输入 **CON**(对于控制台)通知连接程序在屏幕上显示映像, 以便可以立即观察错误。
- **Librayfile** 用于库选项。

如果目标文件和源文件具有相同的名字, 那就不需要重复它, 对驱动器号或路径的引用足以指明文件存储的要求。下面的例子连接目标文件 **A05ASM1.OBJ**。连接程序在驱动器 **n:** 上写 **.EXE** 文件, 显示映像(**CON**), 并且忽略库选项: **LINK n:A05ASM1, n:, CON**。

假如源程序是按 **.COM** 要求写的, 则连接程序显示信息“Warning: No Stack Segment”(“警告: 没有堆栈段”)。

为了把一个以上的目标文件连接成可执行模块, 把它们像这样组合在一行里: **LINK PROCA+PROGB+PROGC**。

如果打算 Code View, 需要使用汇编程序的 **/ZI** 命令行选项。为了连接, 使用 **/CO** 选项作为 **LINK/CO** 文件名。

把 **MASM 5.1** 的目标文件转换为 **.COM** 程序。EXE2BIN 程序把由 **MASM** 产生的 **.EXE** 模块转换成 **.COM** 模块, 源程序必须是按 **.COM** 的要求编码的。键入以下命令:

```
EXE2BIN filename, filename.COM
```

第一个操作数总是引用 **.EXE** 文件, 所以不用编写 **.EXE** 扩展名。第二个操作数可以是任何带 **.COM** 扩展名的有效文件名。如果遗漏了 **.COM** 扩展名, **EXE2BIN** 便假定是 **.BIN**, 为了执行该程序, 其后必须改名为 **.COM**。

E.3 BORLAND TURBO 汇编程序(TASM)

Turbo 汇编程序可以在一个命令行里汇编多个文件，每个文件都有自己的选项。也可以使用通配符(*和?)。为了汇编在当前目录中的所有源程序，键入 TASM *；为了汇编所有名为 PROG1.ASM, PROG2.ASM 等等源程序，键入 TASM PROG?。可以键入文件名组，每组用一个加号(+)分隔。下面的命令汇编带/L 选项的 PROG1 和 PROG2 以及带/Z 选项的 PROG3：

```
TASM /L PROG1 PROG2+ /Z PROG3
```

/L 选项使 TASM 产生列表(.LST)文件，而/Z 使之显示出错的源程序行。键入不带命令行的 TCREF 时，显示命令的通用格式和它的选项说明。标准方式还有许多附加的特性。Borland 提供了另外两个汇编程序版本：TASMx 和 TASM32，它们用于保护方式 F。

交叉引用文件。XRF 文件是用于产生程序标号，符号，以及变量的交叉引用列表的。使用 TCREF 把列表转换成排序的交叉引用文件：

```
TCREF xreffile, reffile
```

有关 TCREF 命令的规则类似于 MASM 5.1 CREF 的那些规则。下面的例子是在驱动器 C 上写入一个名为 ASMPROG.REF 的交叉引用文件：TCREF C:ASMPROG, C:。

连接。连接 TASM 程序的命令行是

```
TLINK ob jfile, exe file[, mapfile][, libraryfile]
```

TLINK 命令的规则和 MASM 5.1 LINK 的那些规则类似。

把 Turbo 目标文件转换成.COM 程序。TLINK 允许把目标程序直接转换成.COM 格式，但源程序必须是按.COM 的要求编码的。使用/T 选项：

```
TLINK /T ob jfile, comfile, CON
```

调试选项。如果要使用 TurboDebugger，可以使用汇编程序的/ZI 命令行选项。对于连接，使用/V 选项，如 TLINK /V filename。

E.4 汇编程序表

在汇编程序.LST 列表后面的是段与组表以及符号表。

段与组表(Segments and Groups Table)。这个表有下面的一些标题：

```
Name Length Align Combine Class
```

- name 列按字母顺序列出所有段和组的名字。
- length 列给出每个段的十六进制的大小。
- align 列给出对准的类型，比如 BYTE, WORD 或 PARA。
- combine 列列出所定义的组合类型，比如堆栈是 STACK，没有编写类型时是 NONE，

外部定义是 PUBLIC, AT 类型是十六进制地址。

- class 列列出段的类别名, 与在 SEGMENT 语句中所编写的相同。

符号表(Symbol Table)。符号表有下面的一些标题:

Name	Type	Value	Attribute
------	------	-------	-----------

- name 列按字母顺序列出所有已定义项的名字。

- type 列给出类型, 如下所示:

L NEAR 或 L FAR: 近的或远的标号

N PROC 或 F PROC: 近的或远的过程

BYTE, WORD, DWORD, FWORD, QWORD, TBYTE: 数据项

ALIAS: 符号的别名(或绰号)

NUMBER: 绝对标号

OPCODE: 指令操作数

TEXT: 文本

- value 列给出名字、标号及过程距离段起点的十六进制偏移值。

- attribute 列列出符号的属性, 包括它所在的段和长度。

MASM 6.1 还列出了过程, 参数, 以及局部表, 这些表标明了它们的名字与属性。

键盘扫描码和 ASCII 码

在下面的列表中，键被分成几类。对每一类都列出了普通键(不与其他键组合)的格式以及 Shift、Ctrl 和 Alt 键组合的格式。在每一列开头的“Normal,”“Shift,”“Ctrl,”和“Alt”下面，有两个 16 进制的字节，其值与键盘操作将它们发送给 AH 和 AL 寄存器的值相同。例如：按下字符“a”时，BIOS 会将扫描码 1EH 发送到 AH 中，并且将 ASCII 字符 61H 发送到 AL 中。当按下 shift 键输入大写字母“A”时，键盘分别传送 1EH 和 41H。扫描码为 85H 及其以上值的键对应扩展键盘。

字 母	NORMAL		SHIFT		CTRL		ALT	
a 和 A	1E	61	1E	41	1E	01	1E	00
b 和 B	30	62	30	42	30	02	30	00
c 和 C	2E	63	2E	43	2E	03	2E	00
d 和 D	20	64	20	44	20	04	20	00
e 和 E	12	65	12	45	12	05	12	00
f 和 F	21	66	21	46	21	06	21	00
g 和 G	22	67	22	47	22	07	22	00
h 和 H	23	68	23	48	23	08	23	00
i 和 I	17	69	17	49	17	09	17	00
j 和 J	24	6A	24	4A	24	0A	24	00
k 和 K	25	6B	25	4B	25	0B	25	00
l 和 L	26	6C	26	4C	26	0C	26	00
m 和 M	32	6D	32	4D	32	0D	32	00
n 和 N	31	6E	31	4E	31	0E	31	00
o 和 O	18	6F	18	4F	18	0F	18	00
p 和 P	19	70	19	50	19	10	19	00
q 和 Q	10	71	10	51	10	11	10	00
r 和 R	13	72	13	52	13	12	13	00
s 和 S	1F	73	1F	53	1F	13	1F	00
t 和 T	14	74	14	54	14	14	14	00
u 和 U	16	75	16	55	16	15	16	00
v 和 V	2F	76	2F	56	2F	16	2F	00

续表

字母	NORMAL		SHIFT		CTRL		ALT	
w 和 W	11	77	11	57	11	17	11	00
x 和 X	2D	78	2D	58	2D	18	2D	00
y 和 Y	15	79	15	59	15	19	15	00
z 和 Z	2C	7A	2C	5C	2C	1A	2C	00
Spacebar	39	20	39	20	39	20	39	20

功能键	NORMAL		SHIFT		CTRL		ALT	
F1	3B	00	54	00	5E	00	68	00
F2	3C	00	55	00	5F	00	69	00
F3	3D	00	56	00	60	00	6A	00
F4	3E	00	57	00	61	00	6B	00
F5	3F	00	58	00	62	00	6C	00
F6	40	00	59	00	63	00	6D	00
F7	41	00	5A	00	64	00	6E	00
F8	42	00	5B	00	65	00	6F	00
F9	43	00	5C	00	66	00	70	00
F10	44	00	5D	00	67	00	71	00
F11	85	00	87	00	89	00	8B	00
F12	86	00	88	00	8A	00	8C	00

数字键区	NORMAL		SHIFT		CTRL		ALT	
Ins and 0	52	00	52	30	92	00		
End and 1	4F	00	4F	31	75	00	00	01
下箭头(↓)and 2	50	00	50	32	91	00	00	02
PgDn and 3	51	00	51	33	76	00	00	03
左箭头(←)and 4	4B	00	4B	34	73	00	00	04
5 (键区)	4C	00	4C	35	8F	00	00	05
右箭头(→)and 6	4D	00	4D	36	74	00	00	06
Home and 7	47	00	47	37	77	00	00	07
上箭头(↑)and 8	48	00	48	38	8D	00	00	08
PgUp and 9	49	00	49	39	84	00	00	09
+ (灰键)	4E	00	4E	2B	90	00	4E	00
- (灰键)	4A	00	4A	2D	8E	00	4A	00
Del and.	53	00	53	2E	93	00		
* (灰键)	37	2A	37	2A	96	00	37	00

上排键	NORMAL		SHIFT		CTRL		ALT	
` 和 ~	29	60	29	7E			29	00
1 和 !	02	31	02	21			78	00
2 和 @	03	32	03	40	03	00	79	00
3 和 #	04	33	04	23			7A	00
4 和 \$	05	34	05	24			7B	00
5 和 %	06	35	06	25			7C	00
6 和 ^	07	36	07	5E	07	1E	7D	00
7 和 &	08	37	08	26			7E	00
8 和 *	09	38	09	2A			7F	00
9 和 (0A	39	0A	38			80	00
0 和)	0B	30	0B	29			81	00
- 和 _	0C	2D	0C	5F	0C	1F	82	00
= 和 +	0D	3D	0D	2B			83	00

操作键	NORMAL		SHIFT		CTRL		ALT	
Esc	01	1B	01	1B	01	1B	01	00
Backspace	0E	08	0E	08	0E	7F	0E	00
Tab	0F	09	0F	00	94	00	A5	00
Enter	1C	0D	1C	0D	1C	0A	1C	00

标点符号	NORMAL		SHIFT		CTRL		ALT	
[和 {	1A	5B	1A	7B	1A	1B	1A	00
] 和 }	1B	5D	1B	7D	1B	1D	1B	00
; 和 :	27	3B	27	3A			27	00
' 和 "	28	27	28	22			28	00
\ 和	2B	5C	2B	7C	2B	1C	2B	00
, 和 <	33	2C	33	3C			33	00
. 和 >	34	2E	34	3E			34	00
/ 和 ?	35	2F	35	3F			35	00

以下为扩展键盘上的复制键（第一个输入键（斜线）为 ASCII 字符，其他为命令键）：

键	NORMAL		SHIFT		CTRL		ALT	
斜线(/)	E0	2F	E0	2F	95	00	A4	00
Del	53	E0	53	E0	93	E0	A3	00
下箭头(↓)	50	E0	50	E0	91	E0	A0	00
End	4F	E0	4F	E0	75	E0	9F	00

续表

键	NORMAL		SHIFT		CTRL		ALT	
Enter	E0	0D	E0	0D	E0	0A	A6	00
Home	47	E0	47	E0	77	E0	97	00
Ins	52	E0	52	E0	92	E0	A2	00
左箭头(←)	4B	E0	4B	E0	73	E0	9B	00
PageDown	51	E0	51	E0	76	E0	A1	00
PageUp	49	E0	49	E0	84	E0	99	00
右箭头(→)	4D	E0	4D	E0	74	E0	9D	00
上箭头(↑)	48	E0	48	E0	8D	E0	98	00

尽管 BIOS 不会将其送入键盘缓冲区，以下的控制键也有定义好的扫描码：

CapsLock	3A
Shift(右)	36
NumLock	45
Alt	38
ScrollLock	46
Ctrl	1D
Shift(左)	2A
PrtScreen	37